НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»
ФІЗИКО-ТЕХНІЧНИЙ ІНСТИТУТ
КАФЕДРА МАТЕМАТИЧНИХ МЕТОДІВ ЗАХИСТУ ІНФОРМАЦІЇ

«На правах рукопису»
УДК __ 519.257 _____

«До захисту допущено»
В.о. завідувача кафедрою
_____ М.М.Савчук
(підпис) (ініціали, прізвище)
“___” _____ 2020р.

Магістерська дисертація

на здобуття ступеня магістра

зі спеціальності 113 Прикладна математика_____
(код і назва)

на тему: Побудова атак збоїв на легкі шифри з ARX-дизайном _____
_____
_____

Виконав (-ла): студент (-ка) __6__ курсу, групи _ФІ-83мн_____
(шифр групи)

Блинов Сергій Юрійович_____ _____
(прізвище, ім’я, по батькові) (підпис)

Керівник к.т.н., доцент кафедри ММЗІ, Яковлєв Сергій Володимирович __
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант _____ _____ _____
(назва розділу) (науковий ступінь, вчене звання, , прізвище, ініціали) (підпис)

Рецензент_____ _____
(посада, науковий ступінь, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2020_року

Національний технічний університет України
«Київський політехнічний інститут
імені Ігоря Сікорського»
Фізико-технічний інститут
Кафедра математичних методів захисту інформації

Рівень вищої освіти: другий (магістерський) за освітньо–професійною програмою

Спеціальність: 113 «Прикладна математика»

ЗАТВЕРДЖУЮ
В.о. завідувача кафедрою
_____ М.М.Савчук
(підпис) (ініціали, прізвище)
«___»_____20_ р.

ЗАВДАННЯ
на магістерську дисертацію студенту

Блинов Сергій Юрійович _____
(прізвище, ім'я, по батькові)

1. Тема дисертації : Побудова атака збоїв на легкі шифри з ARX-дизайном
_____
_____ ,
науковий керівник дисертації : Яковлєв Сергій Володимирович, к.т.н., доцент кафедри ММЗІ _____ ,
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від _____р. № _____

2. Термін подання студентом дисертації _____

3. Об'єкт дослідження : інформаційні процеси в системах криптографічного захисту _____
_____

4. Предмет дослідження (Вхідні дані – для магістерської дисертації за освітньо–професійною програмою):
статистичні атаки збоїв на ARX-криптосистеми _____
_____
_____

5. Перелік завдань, які потрібно розробити _____
1) запропонувати атаку збоїв на шифр SCHWAEMM _____
2) експериментально дослідити ефективність запропонованої атаки _____
3) сформулювати рекомендації для захисту реалізацій від подібних атак_____

_____

_____

6. Орієнтовний перелік ілюстративного матеріалу _____

_____

_____

7. Орієнтовний перелік публікацій _____

_____

_____

8. Консультанти розділів дисертації*

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

9. Дата видачі завдання _____

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання етапів магістерської дисертації	Примітка
1	Пошук і аналіз теоретичних матеріалів	1.03.20	
2	Побудова атаки збоїв	1.04.20	
3	Експериментальна перевірка, аналіз отриманих результатів	1.05.20	
4	Завершення оформлення роботи	11.05.20	

Студент _____ Блинов С.Ю.
(підпис) (ініціали, прізвище)

Науковий керівник дисертації _____ ___Яковлєв С.В._____
(підпис) (ініціали, прізвище)

* Консультантом не може бути зазначено наукового керівника магістерської дисертації.

РЕФЕРАТ

Кваліфікаційна робота містить: 58 стор., 9 рисунків, 10 таблиць і алгоритмів, 18 джерел.

Об'єкт дослідження: інформаційні процеси в системах криптографічного захисту

Предметом дослідження є статистичні атаки збоїв на ARX-криптосистеми.

Метою дослідження є вдосконалення методів криптоаналізу реалізацій криптографічних примітивів у певних обчислювальних середовищах.

При розв'язанні поставлених завдань використовувались наступні *методи дослідження*: методи лінійної та абстрактної алгебри, теорії імовірностей, математичної статистики, комбінаторного аналізу, методи комп'ютерного та статистичного моделювання.

У даній роботі проаналізовані старі та нові атаки збоїв на ARX-криптосистеми, та запропонована атака на процедуру Sparkle у шифрі Schwaemm, в ході дослідження вдалось зменшити перебір ключа з 2^{192} до 2^{64}, що дозволяє при проведенні атаки 3 рази відновити весь ключ шифрування.

SCHWAEMM, SPARKLE, ARX-КРИПТОСИСТЕМИ

ABSTRACT

Qualification work contains: 58 pages, 9 figures, 10 tables and algorithms, 18 sources.

Object of research: information processes in cryptographic protection systems

The subject of the study are statistical crash attacks on ARX cryptosystems.

The purpose of the study is to improve the methods of cryptanalysis of implementations of cryptographic primitives in certain computing environments.

The following *research methods* were used to solve the tasks: methods of linear and abstract algebra, probability theory, mathematical statistics, combinatorial analysis, methods of computer and statistical modeling.

This paper analyzes old and new crash attacks on ARX cryptosystems, and proposes an attack on the Sparkle procedure in the Schwaemm cipher. In the course of the research it was possible to reduce the key search from 2^{192} to 2^{64}, thus executing attack 3 times allows recovering of the whole encryption key.

SCHWAEMM, SPARKLE, ARX CRYPTOSYSTEMS

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

F_2^n – множина бітових рядків довжини n

F_2^* – множина бітових рядків довільної довжини

$\oplus$ – операція побітового додавання

$+$ – додавання за модулем 2^{32}

$||$ – конкатенація бітових рядків

$x \lll s$ слово x циклічно зсувається вліво на s біт

$x \ggg s$ слово x циклічно зсувається вправо на s біт

$x \ll s$ слово x зсувається вліво на s біт

$x \gg s$ слово x зсувається вправо на s біт

ϵ – пустий бінарний рядок

Слово – елемент F_2^{32}

Гілка – пара двох слів (x, y)

IoT – Internet of Things

ARX – Add-Rotation-XOR

ВСТУП

Актуальність З появою Інтернету речей (IoT) безліч пристроїв приєднуються один до одного для обміну інформацією. Ця інформація повинна бути захищена. Симетрична криптографія може гарантувати, що дані, якими обмінюються ці пристрої, залишаються конфіденційними, автентифікованими, та що їх не було підроблено. Оскільки такі об'єкти мають невелику обчислювальну потужність, алгоритими повинні використовувати якомога менше ресурсів. Щоб задовільнити ці потреби, NIST закликав розробити автентифіковані шифри та геш-функції, що забезпечують достатній рівень безпеки при мінімальних витратах ресурів [1]. У даній роботі розглядається алгоритм Schwaemm, побудований з використанням сімейства перестановок Sparkle [3], що є кандидатом на стандарт легкої криптографії. Автентифікований шифр Schwaemm забезпечує конфіденційність відкритого тексту, а також цілісність та автентифікацію для відкритого тексту та для додаткових загальнодоступних даних. Дослідження стійкості цього шифру є відкритим питанням у криптографічнй спільноті, поки йде другий раунд проекту Lightweight Cryptography Standardization.

Метою дослідження є застосування методу криптоаналізу реалізацій криптографічних примітивів у певних обчислювальних середовищах.

Для досягнення мети необхідно виконати такі **завдання**:

1) проаналізувати існуючі методи побудови атак збоїв на симетричні блокові шифри;

2) запропонувати атаку збоїв на шифр SCHWAEMM;

3) експериментально дослідити ефективність запропонованої атаки;

4) сформулювати рекомендації для захисту реалізацій від подібних атак.

Об'єкт дослідження: інформаційні процеси в системах криптографічного захисту

Предметом дослідження є статистичні атаки збоїв на ARX-криптосистеми.

При розв'язанні поставлених завдань використовувались наступні *методи дослідження*: методи лінійної та абстрактної алгебри, теорії імовірностей, математичної статистики, комбінаторного аналізу, методи комп'ютерного та статистичного моделювання.

Наукова новизна: Вперше запропоновано атаку збоїв на шифр Schwaemm у моделі випадкових фіксованих збоїв у 32-бітних словах.

Практичне значення: Результати даної роботи можуть використовуватись для побудови надійних засобів криптографічного захисту інформації.

Апробація результатів та публікації.

– XVIII Всеукраїнській науково-практичній коференції студентів,аспірантів та молодих вчених «Теоретичні та прикладні проблеми фізики, математики та інформатики» (12 - 13 травня, 2020, Київ)

1 ВІДОМІ АТАКИ ЗБОЇВ НА СИМЕТРИЧНІ ARX-КРИПТОСИСТЕМИ

Спочатку ми розглянемо існуючи методи атак збоїв на алгоритми на базі схеми Фейстеля, зокрема, на DES. Ці атаки використовують апарат математичної статистики; на одержаному під час аналізу матеріалі фактично будуються розпізнавачі для правильних ключів.

1.1 Атаки на шифри типу Фейстеля

Схема Фейстеля виконує шифрування блоку даних ітеративним чином; під час кожної ітерації половина блоку замішується із ключем, піддається деякому нелінійному перетворенню (перемішуванню біт) та, можливо, додатковим перетворенням.

Надалі будемо користуватись такою нотацією. Вхідний блок з $2n$ біт розбивається на частини (L_0, R_0) по n біт кожен, після чого послідовно обчислюються такі змінні:

$$L_{m+1} = R_m$$

$$R_{m+1} = L_m \oplus F(R_m, k_{m+1}),$$

де $F(.)$ – раундова функція, k_m – раундовий ключ, що обчислюється деякою функцією. Шифротекстом вважається блок (R_r, L_r), де r – кількість раундів.

Перші атаки збоїв на фейстелівські шифри, зокрема, на DES, були запропоновані ще у 1996 році Біхамом та Шаміром. В подальшому вони були покращені Аккаром у 2004 році.

1.2 Атаки Біхама-Шаміра та Аккара

Атаки Біхама-Шаміра на DES відносяться до класу диференціальних атак збоїв, що викривають декілька біт ключа (зокрема, викривається останній раундовий ключ). При проведенні атаки використовується той факт, що раундова функція шифру DES має блокову структуру, тобто вхідні дані розбиваються на 8 блоків, кожен з яких оброблюється незалежно, тому раундову функцію можна представити у вигляді набору $F = (f_1...f_8)$, де кожна f_i є 4-бітовим перетворенням (зауважимо, що під час обробки вхідні дані розширюються до 6-ти біт)

Сутність атаки полягає у внесенні помилки в оброблюваний блок даних безпосередньо перед останнім раундом (для DES це 16-й раунд) та порівнянні одержаного «збитого» шифротексту із оригінальним. Нехай помилка була внесена у блок R_{15}; позначимо збитий блок через $\tilde{R}_{15}$. Тоді маємо:

$$\tilde{L}_{16} = \tilde{R}_{15}, \tilde{R}_{16} = L_{15} \oplus F(\tilde{R}_{15}, k_{16})$$

та відповідний диференціал:

$$(L_{16} \oplus \tilde{L}_{16}, R_{16} \oplus \tilde{R}_{16}) = (L_{16} \oplus \tilde{L}_{16}, F(R_{15}, K_{16}) \oplus F(\tilde{R}_{15}, K_{16}))$$

Звідси маємо перевірочні співвідношення:

$f_i(R_{15},k_{16,i}) \oplus f_i(\tilde{R}_{15},k_{16,i}) = (R_{16} \oplus \tilde{R}_{16})_i$, де справа беруться i-ті 4-бітові блоки, а $k_{16,i}$ - i-тий 6-бітний блок ключа. Отже, для збою у блоці R_{15} атака будується так:

1) Виконати збій та одержати відповідний «збитий» шифротекст.

2) Для кожного i припустити значення $k_{16,i}$ та перевірити його через перевірочне співвідношення $f_i(R_{15},k_{16,i}) \oplus f_i(\tilde{R}_{15},k_{16,i}) = (R_{16} \oplus \tilde{R}_{16})_i$.

3) Якщо перевірка не проходить, вважати зроблене припущення

невірним.

4) Якщо перевірка пройшла для декількох значень $k_{16,i}$, повторити процедуру для цих значень, але з іншим збоєм.

Не завжди можна внести помилку саме у блок R_{15}. Частіше можна внести помилку у виконання 15-го раунду, що приведе до випадкової зміни всього оброблюваного блоку в цілому. В такому випадку маємо:

$$\tilde{L}_{16} = \tilde{R}_{15}, \tilde{R}_{16} = \tilde{L}_{15} \oplus F(\tilde{R}_{15}, k_{16}),$$

відповідний диференціал:

$$(L_{16} \oplus \tilde{L}_{16}, R_{16} \oplus \tilde{R}_{16}) = (L_{16} \oplus \tilde{L}_{16}, L_{15} \oplus F(R_{15}, K_{16}) \oplus \tilde{L}_{15} \oplus F(\tilde{R}_{15}, K_{16}))$$

та перевірочне співвідношення

$$f_i(R_{15}, k_{16,i}) \oplus f_i(\tilde{R}_{15}, k_{16,i}) = (R_{16} \oplus \tilde{R}_{16})_i \oplus (L_{15} \oplus \tilde{L}_{15})_i,$$

Проблема полягає в тому, що значення $L_{15} \oplus \tilde{L}_{15}$, взагалі кажучи, є невідомим. Для встановлення цього значення Біхам і Шамір використовували додатковий збій у один біт на 14-му та 15-му раунді. Аккар робив додатковий збій у 13-му раунді, користуючись тим, що $L_{13} \oplus \tilde{L}_{13} = L_{15} \oplus \tilde{L}_{15}$.

Після встановлення значення $L_{15} \oplus \tilde{L}_{15}$ проведення атаки не відрізняється від описаної вище.

1.3 Атаки Рівайна

У 2009 році Матьє Рівайн, відштовхуючись від ідей Біхама-Шаміра та Аккара, запропонував схему атаки збоїв на DES, що використовує збої не лише у останньому раунді, а в довільному центральному раунді схеми Фейстеля.

Основним інструментом оцінювання в атаці Рівайна виступає серія припускаючих функцій $g_i(k) = f_i(\tilde{R}_{15},k) \oplus f_i(R_{15},k) \oplus (R_{16} \oplus \tilde{R}_{16})$ (окрема функція для кожного 4-бітного блоку). Якщо значення змінної k співпадатиме з істинним значенням ключа $k_{16,i}$, то $g_i(k) = (L_{15} \oplus \tilde{L}_{15})_i$; в іншому випадку значення $g_i(k)$ буде рівномірно розподіленим. Отже, якщо величина $(L_{15} \oplus \tilde{L}_{15})_i$ має нерівномірний розподіл, то функція $g_i(k)$ виступає розпізнавачем неправильних ключів.

Таким чином, загальна схема атаки Рівайна виглядає так:

1) Виконати N збоїв та зібрати відповідні N пар «коректний шифротекст – збитий шифротекст».

2) Обчислити значення $g_i(k)$ на зібраних парах та припустимих значеннях k.

3) Перевірити гіпотезу «$g_i(k)$ мають такий саме розподіл, що й $(L_{15} \oplus \tilde{L}_{15})_i$»; якщо гіпотеза виконується, то ми знайшли вірне значення ключа.

Для розрізняння гіпотез Рівайн пропонує два шляхи. У випадку, коли модель проведення збоїв фіксована або дискретна, розподіл величини $(L_{15} \oplus \tilde{L}_{15})_i$ фактично відомий до проведення атаки (його можна обчислити безпосередньо: $p_i(x) = P[L_{15} \oplus \tilde{L}_{15} = x]$), і тоді для розрізнення використовується підхід максимальної правдоподібності із відстанню $d(k) = \sum_{j=1}^{N} log(p_i(g_i^{(j)}(k)))$, де j – номер пари шифротекстів, для якої обчислювались припускаючі функції. В іншому випадку, коли розподіл $(L_{15} \oplus \tilde{L}_{15})_i$ невідомий, для розрізнення використовується квадратична евклідова відстань до рівномірного розподілу: $d(k) = \sum_{x=0}^{15} (\frac{\#\{g_i^{(j)}(k)=x\}}{N} - \frac{1}{16})^2$.

Рівайн розглядає збої двох типів: бітові та байтові (блоками по 8 біт); також має значення метод вибору позиції для внесення збоїв: визначеним чином або випадково. Звідси маємо чотири типи збоїв та дві моделі розрізнювачів, наведені вище. Рівайн наводить значення для кількості збоїв, що потрібні для визначення раундового ключа 16-го раунду DES із імовірністю $> 99\%$, в залежності від раунду, де ці збої

відбулись. Значення вказані у таблиці 1.1. Таким чином, атака Рівайна виявляється ефективною навіть для байтової помилки на останніх семи-восьми раундах DES.

Висновки до розділу 1

Ці атаки можна застосувати до будь-якого шифру, побудованого на базі схеми Фейстеля, зокрема до ARX-блоку Sparkle в автентифікованому шифрі Schwaemm, що й було основною темою мого дослідження. Як можна побачити, ані атаки Біхама-Шаміра, ані атаки Рівайна не використовували внутрішню структуру раундового перетворення DES. Проте функції розпізнавання для шифру Sparkle будуть використовувати внутрішню структуру одного раунду перетворення ARX-блоку Sparkle. Було виявлено, що шифр піддається атакам заміни операцій під час фінального додавання та класичним атакам збоїв конкретних чи випадкових бітів або байтів під час раундового перетворення. Втім, дані атаки відновлюють не власне ключ шифрування, а лише одну з його частин, відповідно до структури ARX-блоку, у шифрі Schwaemm це два слова по 32 біти. Для визначення інших частин ключа та відновлення вихідного ключа шифрування потрібно проводити атаку ще таку кількість разів, на скільки частин був розділений ключ.В нашому випадку відповідно 6 разів,оскільки довжина ключа 192 біти. Повний перебір розміру 2^{64} є занадто великим для практичного проведення атаки на одній пересічній машині, проте для більш потужних пристроїв, таких як суперкомп'ютери, такий перебір не є проблемою і атака стає практичною. Даний підхід вимагає додаткових технік для реалізації плану.

Наявність блокової структури у раундовій функції дуже спрощує аналіз, оскільки для блокової структури збої можна вносити в окремі блоки, що локалізує помилку і дозволяє аналізувати розподіл на меншій кількості даних, що зменшує розмір простору для перебору ключа. В той

Таблиця 1.1 – Кількість збоїв, що необхідна для відновлення раундового ключа 16-го раунду DES, в залежності від номеру раунду, типу збою та моделі розпізнавача

Раунд	Розпізнавач	Бітова помилка		Байтова помилка	
		обрана	випадкова	обрана	випадкова
12	Правдоподібність	7	11	9	17
	Евклідова відстань	14	12	17	21
11	Правдоподібність	11	44	210	460
	Евклідова відстань	30	71	500	820
10	Правдоподібність	290	1500	13400	18500
	Евклідова відстань	940	2700	26400	23400
9	Правдоподібність	$3.4 * 10^5$	$2.2 * 10^7$	$> 10^8$	$> 10^8$
	Евклідова відстань	$1.4 * 10^6$	$> 10^8$	$> 10^8$	$> 10^8$

же час відсутність блокової структури не є перешкодою для наведених атак: схеми статистичного розпізнавання не змінюються. Однак обчислювальна складність в цьому випадку суттєво зростає за рахунок великої кількості значень раундового ключа. Тому для шифрів, раундові функції яких не мають блокової структури, перед проведенням даних атак треба додатково оцінити загальну складність визначення ключа шифрування (чи деякої його частини) у порівнянні із простим перебором.

Захист реалізації шифрів від атак Біхама-Шаміра полягає у дублюванні останніх раундів шифрування (чи тільки останнього, чи ще одного-двох попередніх): дані обробляються в незалежних контурах, після чого йде звірка результатів. Для захисту від атак Рівайна тим самим методом треба дублювати щонайменше останні сім-вісім раундів, що є порівняно посиленим навантаженням на реалізацію.

2 АНАЛІЗ ОСТАННЬОЇ АТАКИ ЗБОЇВ НА ШИФР ASCON

Автентифіковані шифри схильні до криптографії із секретним ключем, оскільки такий підхід поєднує конфіденційність, цілісність та автентифікацію в одному алгоритмі та пропонує потенційну ефективність над використанням окремих блокових шифрів і хешів. Сучасні криптографічні конкурси та процеси стандартизації намагаються брати до уваги усі слабкі місця автентифікованих шифрів, щоб врахувати вразливості реалізації, такі як атаки помилок. У цьому розділі буде розглянуто атаки помилок на шифр Ascon, обраний CAESAR, як найкращий для використання серед легких шифрів, а саме техніку аналізу статистично неефективних збоїв (statistical ineffective fault analysis, SIFA) із застосуванням подвійних збоїв і розділення ключа. Збої вносяться у два обрані S-блоки для кожного шифрування під час останнього раунду перестановки на стадії фіналізації Ascon. Корректні значення тегів у результаті введення неефективних збоїв потім використовуються для аналізу ключових гіпотез. Складність методу нападу - це компроміс між розміром простору пошуку ключових гіпотез та кількістю подвійних збоїв. Достатня кількість значень коректних тегів, необхідна для відновлення підмножини ключа, залежить від упередженості розподілу збоїв. Експерименти виконувались над деякою імплементацією Ascon, щоб показати, що між 12,5 і 2500 коректних значень тегів (тобто неефективних збоїв) достатньо для відновлення ключа від сильно упереджених до більш рівномірних розподілів збоїв відповідно.

2.1 Опис подвійної атаки збоїв на прикладі шифру Ascon

У зв'язку зі зростанням Інтернету речей (IoT) та збільшенням використання пристроїв з обмеженими ресурсами реалізація безпечних легких шифрів стала критичною. Звичайні протоколи безпеки поєднують алгоритми шифрування, наприклад, блокові або потокові шифри, алгоритми з кодом автентифікації повідомлення (MAC), наприклад функції з ключем-хешем або універсальне хешування, для відповідності вимогам безпеки. Ці протоколи можуть накладати великі накладні витрати, за пам'яттю, потужністю та пропускною здатністю на легкі пристрої. Навпаки, автентифіковані шифри надають конфіденційність інформації, цілісність даних та автентифікацію в одному алгоритмі, який може полегшити реалізацію сервісів безпеки з використанням меншої кількості ресурсів. Сучасні криптографічні конкурси, такі як Конкурс автентифікованого шифрування: безпека, застосовність та надійність (CAESAR), оцінювали автентифіковані шифри на гарантії безпеки та вразливості імплементації для різних випадків використання, включаючи легкі програми. Національний інститут стандартів і технологій США (NIST) також розпочав багаторічний процес стандартизації для легкої криптографії (LWC), в якій автентифіковані шифри оцінюються для ефективної реалізації на платформах з обмеженими ресурсами та простоти включення контрзаходів проти атак за побічними каналами. Аутентифікований шифр Ascon був обраний фіналістом у березні 2018 року, і був оголошений комітетом CAESAR як найкращий вибір для впровадження у легкі пристрої в лютому 2019 року [1]. Збалансований дизайн підходить для імплементації у легких пристроях як на апаратному, так і програмному рівні, та забезпечує ефективне впровадження контрзаходів проти аналізу за побічними каналами (SCA) [2]. Хоча добре розроблені криптографічні алгоритми надійні проти криптоаналізу або атак грубого перебору, фізичні криптографічні

реалізації можуть втрачати важливу інформацію, яка може експлуатуватися при атаках за побічними каналами. Диференціальний аналіз енергії (DPA) - це пасивна методика SCA, яка експлуатує співвідношення між обробленими даними та енергоспоживанням або електромагнітне поле під час роботи пристрою для отримання частини або цілого секрету [3],[4]. Уцьому розділі фокус йде на аналіз збоїв (FA), що є потужною технікою SCA, яка отримує секретну інформацію шляхом введення збоїв у платформу, де працює криптографічний алгоритм і аналізуючи вихід алгоритму. Більшість методів FA використовують вихідну інформацію від помилок у проміжних змінних, спричинених введенням збоїв (Рис. 2.1).

Однак статистично неефективний аналіз збоїв (SIFA), представлений у [5], базується на моделі мінімального збою та використовує ймовірність того, що введення збоїв може не викликати помилки - так зване неефективне введення збою. Основне припущення в SIFA полягає в тому, що ймовірність помилки в результаті введення збою залежить від даних. Іншими словами, розподіл ймовірностей неоднаковий або упереджений. В випадку правильної ключової здогадки, ми очікуємо, що спостерігатимемо певний розподіл зібраних даних від неефективних збоїв. Далі представлено методику аналізу збоїв щодо автентифікованого шифру Ascon на основі SIFA. Під час використання SIFA атакувати Ascon єдиною помилкою виявилось непрактичним, завдяки особливим властивостям його шару перемішування та S-блоку, тому запропоновано подвійний збій зі стратегією розділення секретного ключа, яка в поєднанні з SIFA здатна відновити весь секретний ключ. Ми також припускаємо мінімальну модель збоїв SIFA, в якій основним припущенням є залежність даних від розподілу збоїв. Ця атака збоїв також пропонує компроміс між розміром ключового простору для пошуку та кількістю необхідних експериментів з подвійними збоями.

Ascon побудований на основі губки, показаній на рис. 2.2. Стан шифру, позначений S, складається з п'яти 64-бітних слів x_0, x_1, x_2, x_3, x_4.

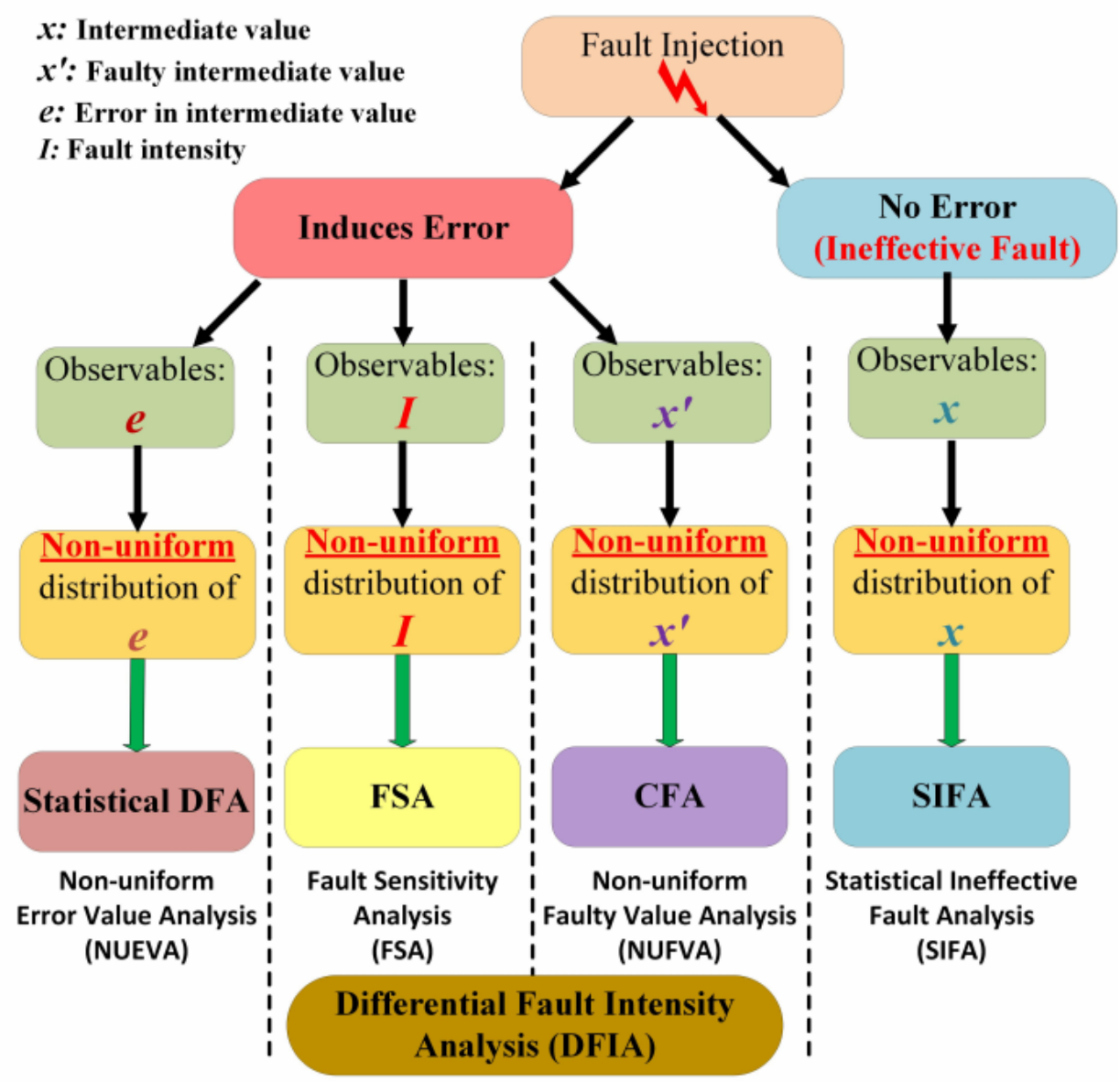

Рисунок 2.1 – Техніки атак збоїв

Відкритий текст і пов'язані з ним дані діляться на блоки по r біт. Кожен блок даних поглинається в перші r біти стану, так звані "rate"біти. Решта прихованих $c = 320 - r$ бітів стану є бітами ємності. Кількість бітів ємності безпосередньо впливає на конфіденційність та межі безпеки аутентифікації шифру на основі губки [6]. Відповідні блоки шифртексту виділяються з конструкції губки Перестановочна функція Ascon складається з додавання константи, шару заміщення (S-блок) і шару перемішування. S-блок - це 5-бітна нелінійна функція. Входи до S-блоку - це біти п'яти слів стану від x_0 до x_4, по одному біту від кожного слова.

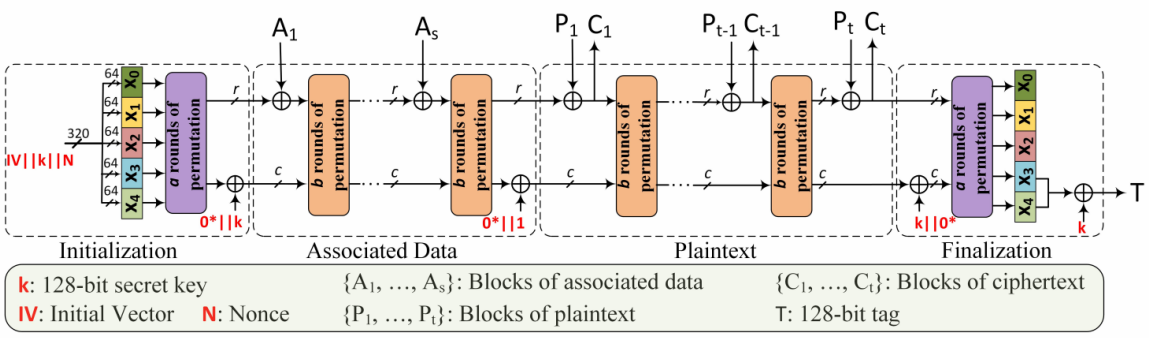

Рисунок 2.2 – Структура шифру Ascon

Концептуально існує 64 S-блоки, відповідно до 64 бітів слів стану.

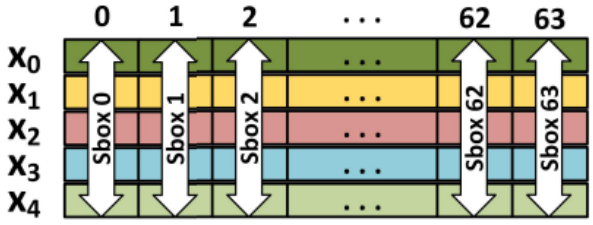

Рисунок 2.3 – Нелінійна операція з використанням S-блоку з одним бітом кожного слова

Функціонування S-блоку в Ascon схематично зображено на рис. 2.3. Оновлений стан на виході операції S-блоку обробляється лінійним шаром перемішування. Функція лінійного перемішування діє на кожне слово $x_i, i = 0, ..., 4$ окремо. Існує п'ять різних лінійних відображень для кожного слова, яке змішує біти в словах. Кожна функція перемішування включає обертальний зсув слова на задані значення, визначені для кожного слова. Лінійне відображення перемішування може бути представлене в матричній формі як

$$\sum_i (x_i) = (L_i x_i) \bmod 2, i = \{0, 1, ..., 4\} \tag{2.1}$$

У цьому зв'язку, x_i - векторне представлення слова стану, і L_i є розрідженою матрицею розмірності $64 * 64$. Множення матриць обчислюється по $\pmod 2$. Перший рядок матриць L_i містить ненульові

елементи в місцях, які представляють значення обертального зсуву відповідної функції перемішування. Як приклад, перший рядок матриці L_3 - вектор з елементами 0, 10 і 17, що дорівнюють 1, а решта - 0. ненульові елементи першого ряду в L_4 також розташовані на 0, 7 та 41. Наступні рядки матриць отримуємо обертанням попереднього рядка праворуч на один елемент.

2.2 Аналіз статистично неефективних збоїв (SIFA)

Диференціальний аналіз збоїв (DFA) аналізує різницю між правильним і збитим шифротекстом, щоб отримати повну або часткову інформацію про секрет [7]-[10]. Статистичний аналіз помилок, з іншого боку, використовує упереджений збій для нападу на криптографічні системи. Спостережувані дані, що використовуються в статистичному аналізі збоїв мають неоднорідний розподіл, як результат введення збою. Статистична атака DFA на AES в [11], використовує нерівномірний розподіл помилки, викликаний проміжним значенням в результаті збою такту. Нерівномірний аналіз значень помилок (NUEVA) та нерівномірний аналіз значень збоїв (NUFVA) покладається на помилку, залежну від даних і розподіл значень збою відповідно [12]. Різні класи статистичного аналізу збоїв, розглянуті вище на рис. 2.1.

Аналіз чутливості до збоїв (FSA) [13] використовує залежну від даних чутливість до збоїв, що визначається як інтенсивність збоїв при якій помилки починають виникати в проміжних змінних внаслідок введення збою. Залежна від даних чутливість до помилок має на увазі, що розподіл чутливості є упередженим. Диференціальний аналіз інтенсивності збоїв (DFIA) розширює FSA шляхом спостереження, що незначна зміна інтенсивності призводить до невеликої зміни помилки, під правильною ключовою гіпотезою [14]. Однак DFIA не використовує інформацію про розподіл; вона зауважує варіацію у вазі Хеммінга несправних значень за різницею інтенсивності збоїв для заданого

відкритого тексту. Атаки збоїв, які містять лише шифротекст (CFA), використовують нерівномірний розподіл значень збоїв x'. У [15] ефект введення збою представлений random-AND моделлю, як $x' = x \odot y$. У цій моделі похибка u розподілена рівномірно. Маємо при рівномірному розподілі x, розподіл збитих значень x' є упередженим, через нелінійність операції AND. Ця властивість є основою методів CFA, якими користуються, щоб отримати секретний ключ шифрування, таких як AES [15], [16] та легких шифрів [17].

Методи аналізу неефективних збоїв припускають, що розподіл x при введенні збоїв неоднаковий. Основні припущення в SIFA полягають в тому, що розподіл збоїв залежить від даних. Ймовірність, що проміжна змінна x, спотворена помилкою через збій, залежить від значення x. Тому розподіл значень x, що не зазнають помилок, при введенні збоїв є упередженим. В атаці SIFA, шифр запускається для декількох вхідних повідомлень, в той час як збій вноситься в проміжну змінну x під час кожного шифрування. Тільки шифротексти, які виникають від введення неефективних збоїв, використовуються в SIFA. Використовуючи зібрані шифротексти, значення x обчислюється, використовуючи зворотну роботу шифру та ключову здогадку. Якщо ключова здогадка вірна, x розподілена так, як очікувалося. Однак якщо ключова здогадка є невірною, тоді значення розподіляються випадковим чином.

2.3 Запропонована атака збоїв

Запропонована методологія аналізу збоїв складається з SIFA у поєднанні з введенням збоїв у вибраній парі S-блоків. А стратегія розділення ключа використовується для зменшення ключового простору для пошуку. Ми приймаємо значення на виході атакованих S-блоків як проміжну змінну, в той час як ціль введення збою – операції S-блоку в останньому раунді стадії Finalization. Завдяки особливим властивостям його перестановочної функції, SIFA не в змозі атакувати Ascon,

використовуючи введення збою в один S-блок. По-перше, обернена операція з виходу шифру до виходу S-блоку під будь-якою ключовою гіпотезою, є лінійним відображенням один в один. Отже, для будь-якої ключової здогадки, розподіл даних на виході S-блоку розподілений нерівномірно з однаковим зміщенням. По-друге, обернена функція перемішування для слів стану x_3 та x_4 використовує 68 бітів виходу, а також ключ для обчислення бітів входу шару перемішування. Отриманий простір пошуку розміру 2^{68} не є практичним для перебору. Щоб з цим впоратись запропоновано стратегію розділення ключа.

2.3.1 Модель і розподіл збою

У цій моделі збою, нападник здатний вводити збої в будь-яку обрану пару S-блоків останнього раунду стадії Finalization. Це називається експериментом подвійного збою. У цій моделі більше двох S-блоків може зазнати збій. Поки збій викликається у двох обраних S-блоках, атака успішна. Зловмисник може запустити шифр для багатьох відкритих текстів з подвійною помилкою та однаковим значенням ключа. Початковий стан може не бути однаковим для різних шифрувань. Тільки нападник вимагає коректних значень тегів під час введення збою.

У цій моделі збою передбачається, що ймовірність розподілу неефективних збоїв упереджена. Ця модель атаки має менше необхідних умов, ніж багато інших методик диференціального аналізу. Атаку можна здійснити за допомогою простих збоїв такту та/або напруги під час роботи S-блоку в останньому раунді. Через статистичну природу аналізу, навіть ефект шуму можна звести до мінімуму, якщо зібрано достатній обсяг даних. Приклади шумових збоїв включають подвійні збої, які не націлені на вибрані S-блоки та/або останній раунд шифрування. У цих випадках атака буде все-таки успішною, якщо достатньо даних внаслідок бажаного збою будуть доступні.

Нерівномірний розподіл збитих значень або правильних значень під

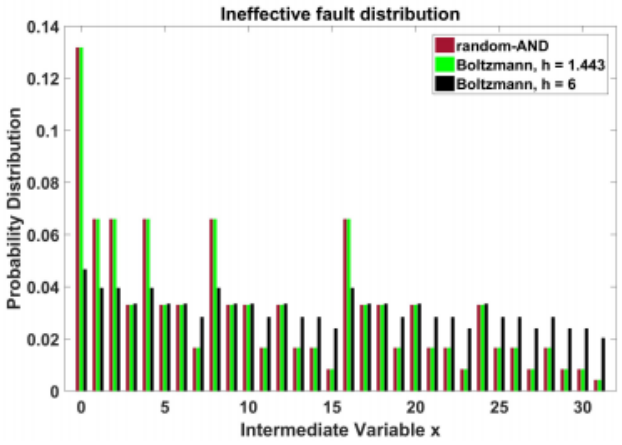

Рисунок 2.4 – Розподіл 5-бітового проміжного значення під неефективним збоєм

неефективними помилками можна пояснити, використовуючи random-AND модель збою. У цій моделі розподіл значень x за неефективного збою можна обчислити як

$$Pr\{x' = x\} = \sum_{u} Pr\{x \odot U = x | U = u\} p_U(u) \tag{2.2}$$

Неефективний розподіл збоїв зображений на рис. 2.4 для рівномірного розподілу похибки u та 5-бітних значень x. У зв'язку з нелінійною властивістю операції AND, x з більшою ймовірністю приймає значення з меншою вагою Хеммінга. З іншого боку, значення x з меншою вагою Хеммінга також з більшою ймовірністю не змінюються після введення збою. Спостереження мотивує вважати, що розподіл Больцмана для значень x під неефективними помилками виглядає як

$$p_X(x) = \frac{e^{-Hw(x)/h}}{\sum_x e^{-Hw(x)/h}} \tag{2.3}$$

У цьому відношенні $Hw(x)$ являє собою вагу Хеммінга x. Причиною моделювання розподілу даних за допомогою функції Больцмана є те, що зміщення розподілу може бути легконалаштовним одним параметром h. Отже, ефективність запропонованої атаки може бути перевірена для різних рівнів розподілів у моделюваннях. Розподіл неефективного збою в

моделі random-AND можна точно отримати, встановивши $h = 1,443$ в розподілі Больцмана. Це значення обчислюється як мінімізація евклідової відстані між розподілом Больцмана та random-AND моделлю. Це спостереження також демонструє, що розподіл збоїв експоненційно залежить від ваги Хеммінга даних у randon-AND моделі. Збільшуючи значення h, зміщення розподілу Больцмана зменшується. В крайньому випадку $h \mapsto \infty$, розподіл Больцмана стає просто рівномірним. Розподіл $h = 6$ також показано на рис. 2.4 для порівняння. Загальна метрика в літературі для вимірювання зсуву розподілу це - квадрат евклідової відстані (SEI). Розглянувши два розподіли $p_a(x)$ та $p_b(x)$, SEI обчислюється як

$$D = \sum_x (p_a(x) - p_b(x))^2 \tag{2.4}$$

Коли $p_a(x)$ – функція маси ймовірності рівномірного розподілу, вищезазначена відстань визначає SEI $p_b(x)$. SEI використовується для вимірювання упередженості розподілу даних так, що мінімальна кількість інформації про розподіл необхідна.

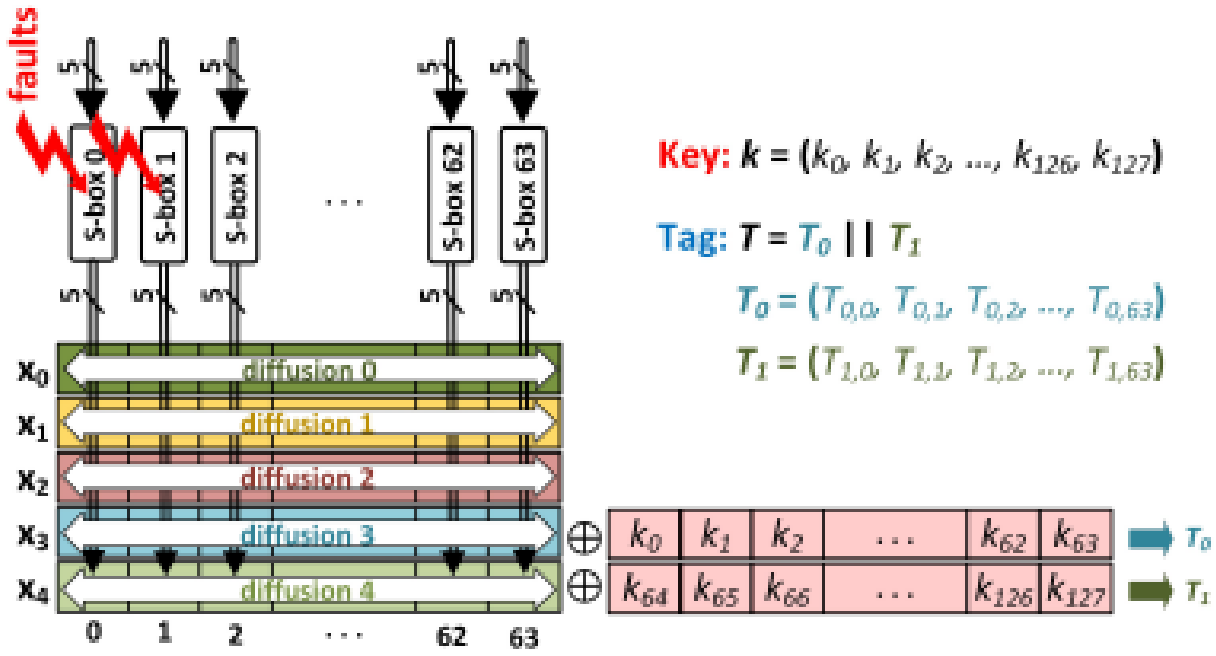

Рисунок 2.5 – Місце для введення подвійного збою

У запропонованій методології атаки збої вносяться у два обраних S-блоки, під час кожного шифрування. Перестановочна операція останнього раунду схематично показана на Рис. 2.5. Після додавання константи, S-блоки працюють зі станом і функцією перемішування, змішують біти в кожному 64-бітному слові x_i. Правильні значення тегів при неефективних збоях зібрані для аналізу. Правильні теги можна ідентифікувати двома способами:

1) автентифікована розшифровка, в якій немає відкритого тексту через невдалу автентифікацію, позначає неправильний тег;

2) ми можемо порівнювати значення тегів з та без внесення помилок.

Дані на виході атакованих S-блоків обчислюються від значень тегів за допомогою зворотньої операції функції перемішування під кожною ключовою гіпотезою. Використовуючи матричне подання, обернені відображення перестановки – це просто обернені матриці L_i^{-1}. Розглядаючи x_i та $\sum_i$ як вхідні та вихідні слова шару перемішування, зворотне перемішування можна представляти як

$$x_i = (L_i^{-1} \sum_i) \bmod 2, i = \{0, 1, ..., 4\} \tag{2.5}$$

Ми представляємо матрицю зворотнього перемішування за її рядками як

$$L_i^{-1} = [l_0^{(i)T}, l_1^{(i)T}, ..., l_{63}^{(i)T}]^T, i = \{0, 1, ..., 4\} \tag{2.6}$$

У цьому відношенні $l_j^{(i)T}$ – це j-й рядок матриці зворотнього перемішування, що відповідає слову стану x_i. Напис T являє собою матрицю транспозиції. Використовуючи це визначення, біти 3 та 4 на виході S-блоку j, позначеного $s_3^{(j)}$ та $s_4^{(j)}$, відповідно, можна обчислити за допомогою рівнянь:

$$s_3^{(j)} = \sum_{r=0}^{63} [(T_{0,r} \oplus k_r) \odot l_{j,r}^{(3)}] \bmod 2 \tag{2.7}$$

$$s_4^{(j)} = \sum_{r=0}^{63} [(T_{1,r} \oplus k_{r+64}) \odot l_{j,r}^{(4)}] \bmod 2 \tag{2.8}$$

У цих рівняннях $T_{0,r}$ та $T_{1,r}$ є r-м бітом першої і другої половинки тегу відповідно (див. рис. 2.5). Також, k_r – r-й біт ключа і $l_{j,r}^{(i)}$, r-й елемент в j-му рядку матриці зворотнього перемішування L_i^{-1}.

Проміжними змінними, які використовуються для статистичного аналізу, є значення бітів 3 і 4 з виходу обраних S-блоків. Якщо проміжну змінну позначають 2-бітним значенням $z^{(j)}$, перший та другий біти у двійковому поданні $z^{(j)}$ є бітами 3 і 4 на виході S-блоку j. Розглянемо випадок, коли Ascon шифрує кілька відкритих текстів з подвійними збоями, які націлюються на S-блоки 0 і 1, на останньому раунді стадії фіналізації, під час кожного шифрування. Припустимо у M шифруваннях збої малоефективні; таким чином, M правильних значень тегів збираються в експерименті з подвійною помилкою. Розподіл зібраних значень на виході S-блоку 0 і 1, тобто $z_1^{(0)}, z_2^{(0)}, ..., z_M^{(0)}, z_1^{(1)}, z_2^{(1)}, ..., z_M^{(1)}$, використовується для оцінки зміщення введених збоїв під ключовою гіпотезою.

Враховуючи функції перемішування Ascon, кожен рядок L_3^{-1} та L_4^{-1} має, відповідно, 33 та 35 ненульових елементів. Це означає, що підмножина ключа, необхідна для обчислення значень одного S-блоку включає $33 + 35 = 68$ біт ключа. Оскільки ми вимагаємо вихідні значення двох S-блоків, розмір ключового підпростору більше, ніж 2^{68}, що досить важко для пряммолінійного грубого перебору. У запропонованому способі 128-бітний ключ розділений на слова довжини w біт, як показано на рис. 2.6, де ми припускаємо що w – степінь 2. Біти 3 і 4 на виході S-блоку j можуть бути обчислені лінійною комбінацією бітів у ключових словіах. Коефіцієнти комбінації визначаються за допомогою j-го рядку матриці зворотнього перемішування L_3^{-1} та L_4^{-1}, як показано на рис. 2.6. Це

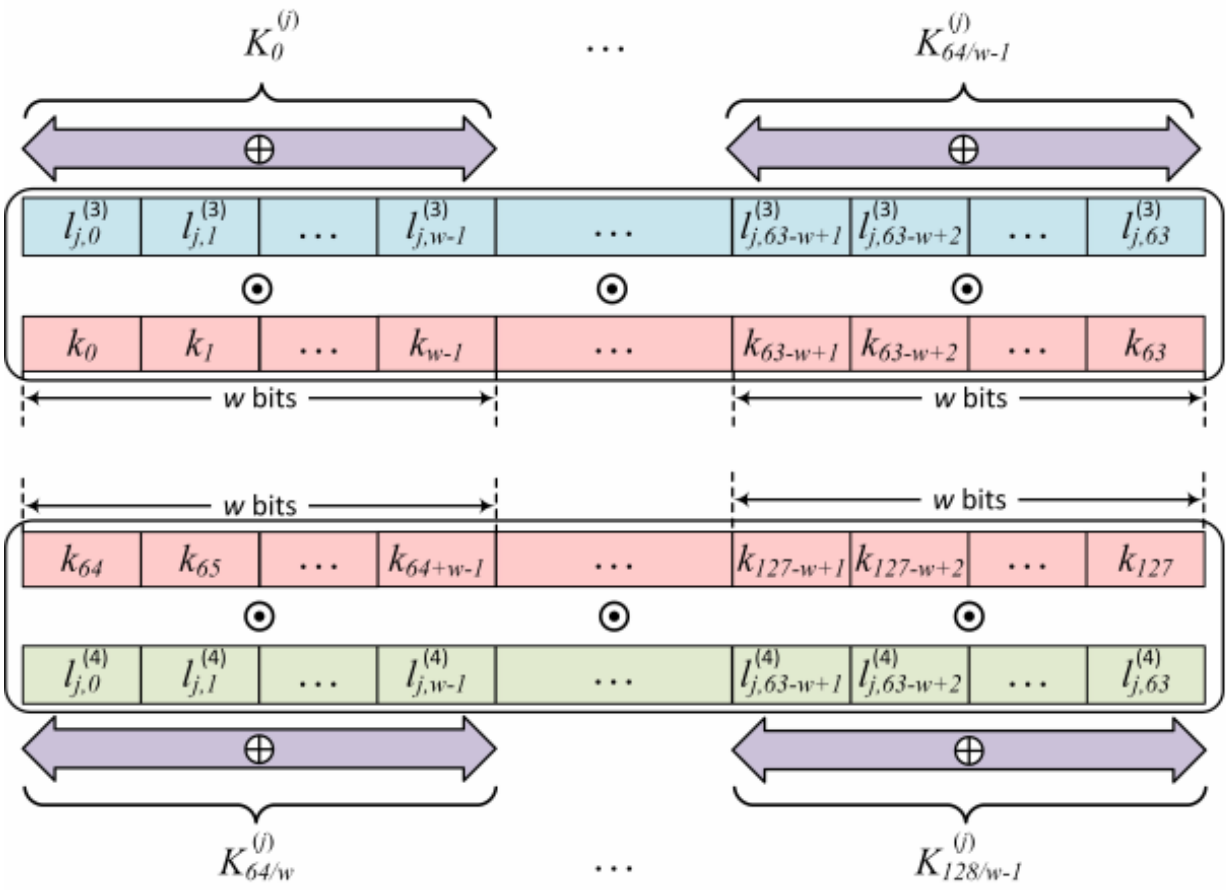

Рисунок 2.6 – Стратегія розділення ключа

можна побачити, переписавши рівняння для бітів 3 і 4 j-ого S-блоку як

$$s_3^{(j)} = \sum_{s=0}^{64/w-1} \Big(\sum_{r=s*w}^{(s+1)*w-1} T_{0,r} \odot l_{j,r}^{(3)} \Big) \oplus K_s^{(j)} \tag{2.9}$$

$$s_4^{(j)} = \sum_{s=0}^{64/w-1} \Big(\sum_{r=s*w}^{(s+1)*w-1} T_{1,r} \odot l_{j,r}^{(4)} \Big) \oplus K_{s+64/w}^{(j)} \tag{2.10}$$

У цих рівняннях ключові бітові комбінації для $s = \{0, 1, ..., 128/w - 1\}$ визначаються як

$$K_s^{(j)} = \sum_{r=s*w}^{(s+1)*w-1} k_r \odot l_{j,r} \tag{2.11}$$

Щоб спростити позначення вводимо конкатенацію $l_j = (l_j^{(3)}, l_j^{(4)})$. Для того, щоб обчислити біти 3 і 4 на виході S-блоку j, нам потрібні лише комбінації бітів ключа замість окремих 68 біт ключа, що

використовуються при обчисленнях. Використовуючи векторне подання, ключовою гіпотезою для S-блоку j буде

$$K^{(j)} = (K_0^{(j)}, K_1^{(j)}, ..., K_{128/w-1}^{(j)}) \tag{2.12}$$

Розглянемо приклад, у якому цілями введення подвійних збоїв є пара S-блоків 0 і 1. Отже, ключовою гіпотезою є вектор $(K^{(0)}, K^{(1)})$. Використовуючи аналіз збоїв, значення $K_s^{(0)}$ та $K_s^{(1)}$ для $s = \{0, 1, ..., 128/w - 1\}$ оцінюються. Кожен $K_s^{(j)}$ – лінійна комбінація w біт ключа всередині слова s. Коли введення збоїв націлене на різні пари S-блоків, виходять різні комбінації w біт ключа отримуються як набір w бінарних рівнянь для w біт ключа у слові. Тоді w біт кожного ключового слова s обчислюється розв'язуванням набору лінійних рівнянь, відповідних даному слову.

2.4 Алгоритм аналіза ключів

Ключова гіпотеза $K_j = (K^{(j)}, K^{(j+1)})$, для двох атакованих S-блоків j та $j + 1$, містять комбінації підмножин ключів. Кожен елемент K_j визначається, коли ключ ділиться на слова по w біт. Розглянемо приклад, в якому ключ ділиться на слова довжиною $w = 32$, тоді ключова гіпотеза K_j включає 8 елементів; розмір простору пошуку таким чином 2^8. Щоб отримати правильні здогадки K_j, SIFA використовується для аналізу даних на виході S-блоку j та $j + 1$. З вичерпним пошуком над усіма 2^8 можливими значеннями K_j, біти 3 і 4 на виході S-блоку j і $j + 1$ рахуються з рівнянь вище. Показник SEI обчислюється для розподілу значень, отриманих для кожної ключової гіпотези. Правильні здогадки для K_j показують максимальний SEI або зміщення розподілу даних. Алгоритм 2.1 детально оцінює поєднання ключових бітів за допомогою атаки SIFA. Припустимо, що 128-бітний ключ розділений на слова по w біт, $w - 1$ експеримент з подвіним збоєм необхідний для пари S-блоків

$(0,1),(1,2),(2,3),...,(w-2,w-1)$. З кожного експеримента із цільовою парою $(j, j+1)$ отримуємо правильні здогадки для комбінацій $(K^{(j)}, K^{(j+1)})$. Далі ключові біти обчислюються за допомогою зібраних ключових комбінацій в алгоритмі аналізу ключових бітів (Алгоритм 2.2).

Algorithm 2.1 Ключова здогадка з SIFA

Проініціалізуємо пусті таблиці S, T
for $j = 0 \to w-2$ **do**
    for $n = 1 \to N$ **do**
        Запустіть шифр для відкритого тексту P_n та введіть помилки у два S-блоки $j, j+1$ на останньому раунді стадії Finalization
        Порівняти отриманий тег з коректним, без збою. Зберегти тег, якщо він правильний
    end for
    Записати зібрані теги як $\{T_1, T_2, ..., T_M\}$
    for $k = 0 \to 2^{256/w} - 1$ **do**
        Встановити $(K^{(j)}, K^{(j+1)})$ одним з можливих $2^{256/w}$ значень
        Ініціалізувати пустий вектор V
        for $m = 1 \to M$ **do**
            Обрахувати біти 3 та 4 на виході S-блоку j, використовуючи $K^{(j+1)}$ та значення тегу T_m.Зберегти отримане значення $z_m^{(j+1)}$ у вектор V
        end for
        Обрахувати SEI для даних, збережених у векторі V
        Зберегти значення SEI у таблиці S
    end for
    Знайти максимальні значення SEI, збережені у таблиці S. Зберегти відповідні значення для комбінацій ключів $(K^{(j)}, K^{(j+1)})$ у рядку j таблиці T.
end for

Треба взяти до уваги, що коректні ключові здогадки $(K^{(j)}, K^{(j+1)})$ отримані в Алгоритмі 2.1 не унікальні. Для кожного значення $K^{(j)}$ є відповідне значення $K^{(j+1)}$ з якого отримуємо такий самий розподіл даних на виході S-блоків j та $j+1$. Таким чином кількість правильних здогадок для комбінації $(K^{(j)}, K^{(j+1)})$ дорівнює $2^{128/w}$. Однак значення $K^{(j)}$ у кожній здогадці унікальне.

Ключові біти можуть бути обчислені зі здогадок $(K^{(j)}, K^{(j+1)}), j = \{0, 1, ..., w-2\}$, використовуючи алгоритм 2.2. Далі за алгоритмом ми беремо одну пару $(K^{(0)}, K^{(1)})$. Серед інших варіантів для $(K^{(1)}, K^{(2)})$, взяти один, з таким самим значенням $K^{(1)}$, як у $(K^{(0)}, K^{(1)})$. Те ж саме для $(K^{(2)}, K^{(3)})$, взяти здогадку з тим самим значенням $K^{(2)}$, як у $(K^{(1)}, K^{(2)})$, і так далі. В кінці матимемо послідовність $\{K^{(0)}, K^{(1)}, ..., K^{(w-1)}\}$. Кожен елемент $K^{(j)}$ містить $128/w$ бінарних

значень, відповідних $128/w$ словам ключа. Для кожного слова s, ключові біти обчислюються рішенням множини лінійних бінарних рівнянь для $j = \{0, 1, ..., w-1\}$. Це призводить до однієї здогадки ключових бітів. Для перевірки коректності ключа, запускаємо шифр з обчисленим ключем. Якщо згенерований шифротекст відповідає очікуваному, ключ коректний. В іншому випадку починаємо з іншої здогадки $(K^{(0)}, K^{(1)})$.

Algorithm 2.2 Аналіз ключових бітів

1: Проініціалізуємо пусту таблицю K
2: **for** $t = 1 \rightarrow 2^{128/w}$ **do**
3: Взяти одну з $2^{128/w}$ здогадок $(K^{(0)}, K^{(1)})$
4: Записати значення $(K^{(0)}$ та $(K^{(1)}$ в рядку t таблиці K
5: **for** $j = 1 \rightarrow w - 2$ **do**
6: З рядка j таблиці T знайти елемент $(K^{(j)}, K^{(j+1)})$, який має такий самий $(K^{(j)}$, як збережений в рядку t таблиці T. Зберегти значення $(K^{(j+1)}$ в рядку t таблиці K
7: **end for**
8: Елементи збережені в рядку t таблиці K зараз $\{K^{(0)}, K^{(1)}, ..., K^{(w-1)}\}$. Далі:
9: **for** $s = 0 \rightarrow 128/w - 1$ **do**
10: Значення формують множину лінійних бінарних рівнянь для бітів ключа у слові. Вирішити множину рівнянь для ключових бітів.
11: **end for**
12: Запустити шифр з обчисленими ключовими бітами для одного відкритого тексту. Порівняти шифротекст з очікуваним. Якщо вони співпадають, повернути ключові біти.
13: **end for**

2.4.1 Результати атаки та достатня кількість даних

Квадратична евклідова відстань (SEI) розподілу даних на виході пари S-блоків $(0, 1)$, на які націлені подвійні збої, порівняно з простором пошуку комбінацій ключа $(K^{(0)}, K^{(1)})$, показано на рис.2.7 для достатньої кількості правильних значень тегів. У наступних експериментах ключ ділиться на два слова довжиною 64 біта. Звідси і простір пошуку $(K^{(0)}, K^{(1)})$ має розмір 16 можливих значень. Під час експерименту максимальний SEI отримується для 4 значень ключового простору. 4 піки відповідають 4 можливим значенням $K^{(0)} = (K_0^{(0)}, K_1^{(0)})$. Усі 4 піки мають однакове значення SEI, оскільки для кожного значення $K^{(0)}$ існує певне значення $K^{(1)}$, що призводить до розподілу даних з однаковим зміщенням, як на виході двох S-блоків. Правильна ключова гіпотеза

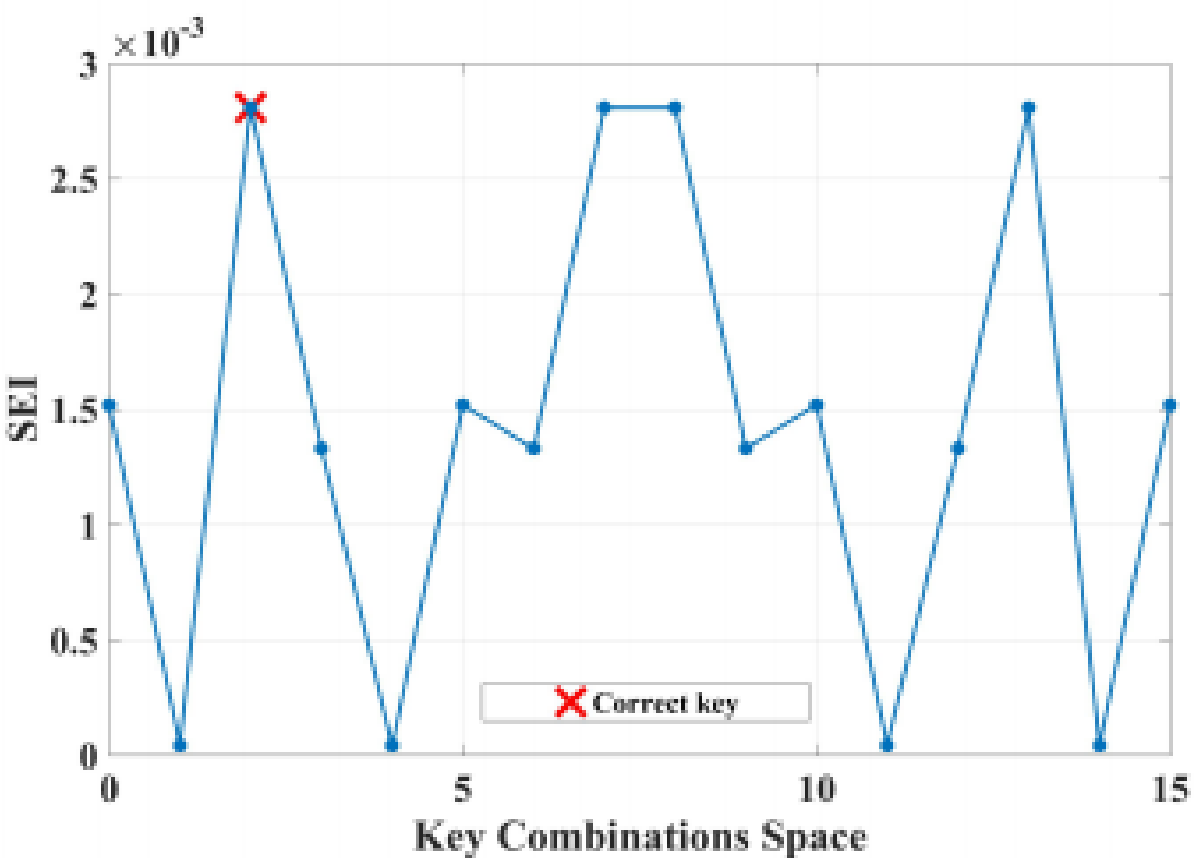

Рисунок 2.7 – SEI для даних з кандидатів ключового простору

позначена на рис. 2.7, яка є одним з 4 значень з піком SEI. Правильна ключова гіпотеза не може бути визначена на цьому етапі, оскільки всі 4 гіпотези мають однаковий пік SEI, незалежно від розміру зібраних даних. Щоб відновити весь ключ, значення SEI та ключовий простір повинні бути обраховані для всіх експериментів з подвійними збоями $(0,1),(1,2),...,(62,63)$. У разі успіху експерименту з активними збоями $(j,j+1)$ і $(j+1,j+2)$ заплутуються крізь S-блок $j+1$, для кожного піку експерименту $(j,j+1)$, існує лише один відповідник експерименту $(j+1,j+2)$. Тому існує всього 4 можливих послідовності $\{K^{(0)},K^{(1)},...,K^{(63)}\}$ і тільки 4 здогадки ключових бітів включають правильний ключ.

Пікові значення SEI включають правильний ключ, якщо достатня кількість даних отримується з кожного експерименту з введенням збоїв. Значення SEI для правильного значення ключа порівнюється з

максимальним SEI для невірних ключових здогадок. Достатній розмір даних для вилучення правильних значень ключа залежить від упередженості розподілу збоїв. У випадку $SEI = 0,001$ розподіл є більш рівномірним, ніж random-AND модель. Отже, кількість текстів, необхідних для виявлення правильного значення ключа виростає щонайменше до 450 – на порядок більше, ніж те, що потрібно для моделі random-AND. Необхідний розмір даних разом з розподілом помилок для кожного експерименту з подвійним збоєм, таким, що SEI обчислених даних з правильними ключовими здогадками більше, ніж з неправильними ключовими здогадками. Ми використовуємо метрику SEI для вимірювання зміщення розподілу збоїв. Ми спостерігаємо, що розмір даних збільшується обернено зміщенню розподілу збоїв. Для заданого набору відкритих текстів може знадобитися більше даних. Одним із обмежень методів статистичного аналізу збоїв є те, що достатній розмір даних, необхідний для відновлення ключа дуже сильно залежить від зміщення розподілу збоїв. Однак зловмисник не знає зміщення в даній реалізації. Просте рішення цього питання – поступово збільшувати розміри даних та відгадати ключ. Якщо значення ключової здогадки не змінюється, це означає, що розмір даних достатній і ключова здогадка правильна.

2.4.2 Збої з шумом

Великою перевагою статистичного аналізу збоїв є його толерантність до шумових збоїв. Зловмисник може мати недостатньо контролю над впровадженням збоїв, наприклад, контролю над часом або проявом збоїв у системі. Далі розглядається вплив шумів щодо запропонованої атаки на Ascon у двох випадках.

У першому випадку зловмисник може вводити збої у неправильні пари S-блоків з вірогідністю $0,5$. Для виявлення потрібного ключа необхідно не менше 1530 значень тегів у випадку $SEI = 0.001$. Для

порівняння необхідно приблизно в 3 рази більше даних для випадку збоїв без шуму.

Подібно до випадку шумових збоїв націлених на неправильні S-блоки з ненульовим значенням ймовірності, коли впровадження збоїв націлене на неправильний раунд з деякою ненульовою ймовірністю, зміщення даних для даної кількості збоїв зменшується. В результаті необхідна кількість даних, необхідних для відновлення правильного ключа, збільшується до 1121, на відміну від необхідного розміру даних у випадку відсутності шумів у збоях (450). Цей випадок може статися в реалізаціях з неефективними контрзаходами, в яких раунди-манекени вставляються в роботу шифру. Отже, нападник не має точного контролю над цільовим раундом для впровадження збою. Однак за рахунок збільшення кількості збоїв, достатній обсяг даних, орієнтований на бажаний раунд набувається.

Висновки до розділу 2

У цьому розділі було продемонстровано методологію статистичного аналізу збоїв та відновлення всього 128-бітного секретного ключа для реалізації аутентифікованого шифру Ascon за допомогою введення подвійних збоїв та стратегії поділу ключа на основі статистичного аналізу неефективних збоїв (SIFA). Ця методологія атаки дозволяє досягти компромісу між кількістю подвійних збоїв та розміром ключового простору пошуку. Діленням ключа на слова по w біт, необхідний $w-1$ експеримент з подвійним збоєм з простором пошуку розміром $2^{256/w}$. Достатня кількість зібраних даних, необхідних для виявлення правильного ключа, у кожному експерименті з подвійним збоєм, залежить від упередженості розподілу збоїв. Для розподілу збоїв, починаючи від дуже упередженого до більш рівномірного, достатня кількість правильних тегів коливається в середньому від $12,5$ до 2500 відповідно. Запропонований спосіб також є толерантним до введення

шумових збоїв. Для зловмисника, націленого на неправильну пару S-блоків або неправильний раунд перестановки з вірогідністю $0,5$, достатня кількість правильних тегів збільшується в 2–3 рази.

Запропонована атака успішно відновлює весь секретний ключ із використанням симуляцій реалізації Ascon на мові програмування C. В в майбутньому ця методика атаки може бути перевірена на різних апаратних реалізаціях шифру, включаючи FPGA, ASIC і вбудовані процесори, використовуючи збій живлення або тактовий збій. Успіх атак оцінюватиметься за допомогою різних реалізацій S-блоків Ascon. Як обговорювалося раніше, більшість звичайних контрзаходів проти атак збоїв не можуть захистити шифр від запропонованої атаки. Це можливо перевірити в реальних технічних реалізаціях. Крім того, протидії неефективним збоям підлягають оцінці.

3 АНАЛІЗ ШИФРУ SCHWAEMM ТА ПОБУДОВА СТАТИСТИЧНОЇ АТАКИ ЗБОЇВ

Далі розглянемо саме автентифікований шифр Schwaemm та спробуємо проаналізувати його стійкість до статистичних атак збоїв. Під час формування розділу були використані наступні матеріали [3],[7],[8],[9],[10],[11].

3.1 ARX-блок Sparkle, Esch та Schwaemm

Ідея полягає у внесенні бітових помилок у процедуру Sparkle, яка використовується у функціях Esch та Schwaemm.

Sparkle – це сімейство криптографічних перестановок, заснованих на ARX дизайні. Його розробили Крістоф Бейерле, Алекс Бірюков, Луан Кардосо Душ Сантуш, Йоганн Гросщадль, Лео Перрін, Олексій Удовенко, Веселін Величков та Цінджу Ванг. Його назва походить від блокового шифру Sparx, з яким Sparkle тісно пов'язаний. Sparkle в основному може бути зрозумілий як екземпляр Sparx з більшим розміром блоку та фіксованим ключем, звідси його назва: SPARx, але Key LEss.

Існує три версії, що відповідають трьом розмірам блоків, а саме Sparkle256, Sparkle384 та Sparkle512. На відміну від більшості ARX конструкцій, розробники гарантують безпеку щодо диференціального та лінійного криптоаналізу завдяки стратегії довгого сліду (LTS). Запропонована структура також зручна для дослідження інших типів атак (інтегральних, неможливих диференціальних, атак збоїв тощо) і, таким чином, дає змогу сперечатися про безпеку цих алгоритмів проти них. LTS – це стратегія, яка вперше була використана для розробки легкого блокового шифру Sparx, представлена на ASIACRYPT 2016. Кілька незалежних дослідницьких команд вже проаналізували цей

алгоритм, і їх результати дають впевненість у цьому дизайнерському підході, яку я буду намагатися перевірити.

Esch та Schwaemm – це криптографічні алгоритми, розроблені таким чином, щоб вони мали легку програму (тобто мали невеликий розмір коду та використовували невеликий обсяг RAM) і все ще досягали високої продуктивності в діапазоні 8, 16 та 32-бітових мікроконтролерів. Esch та Schwaemm також можуть бути оптимізовані зменшенням площі кремнію та енергоспоживання, якщо впроваджені в апаратне забезпечення. Ці схеми побудовані на основі добре зрозумілих принципів, тобто sponge конструкції, заснованої на сімействі перестановок Sparkle.

Esch – це сімейство криптографічних хеш-функцій. Назва розшифровується як Effective Sponge Cheap Hashing. Це також частина назви невеликого містечка на півдні Люксембурга, що знаходиться недалеко від кампусу Люксембургського університету. Esch використовує сімейство перестановок Sparkle в губці.

Ефективність вищезазначених шифрів полягає в наступному:

Невеликий розмір стану

І для Esch, і для Schwaemm характерний порівняно невеликий розмір стану, який становить лише 256 біт для найлегшого екземпляра Schwaemm (досягнення рівня безпеки 120 біт) і 384 біт для найлегшого варіанту Esch. Малий стан – важливий актив для легких криптосистем з кількох причин. Перш за все, розмір стану значною мірою визначає споживання оперативної пам'яті (у разі впровадження програмного забезпечення) та область кремнію (якщо реалізується апаратно) симетричного алгоритму. Крім того, реалізація програмного забезпечення для 8 і 16-бітових мікроконтролерів з невеликим розміром регістра (наприклад, Atmel AVR або TI MSP430) може отримати значні переваги від невеликого розміру стану, оскільки дозволяє великій частці стану перебувати в регістрах, що зменшує кількість операцій завантаження та

збереження. На 32-бітних мікроконтролерах (наприклад, серії ARM Cortex-M) навіть можливо зберегти повний 256-бітний стан в регістрах, тим самим усуваючи майже всі навантаження на ресурси. Можливість утримувати весь стан в регістрах не тільки приносить користь часу виконання, але й забезпечує певний внутрішній захист від атак за побічними каналами.

Надзвичайно легка перестановка

Сімейство перестановок Sparkle – це класичниа ARX криптосистема, яка виконує додавання, зсув та операції XOR на 32-бітних словах. Використання 32-бітного розміру слова забезпечує високу ефективність програмного забезпечення на 8, 16 та 32-бітних платформах; менші розміри слів (наприклад, 16 біт) погіршать продуктивність на 32-бітних платформах, тоді як 64-бітні слова є проблематичними для 8-бітових мікроконтролерів. Кількість зсувів були ретельно вибрані, щоб мінімізувати час виконання та розмір коду на мікроконтролерах, які підтримують лише зсув по одному біту за один раз. Впровадження Sparkle для мікроконтролерів ARM може використовувати їх здатність поєднувати додавання або XOR із зсувом в одну інструкцію із затримкою на один тактовий цикл. З іншої сторони, апаратна реалізація може скористатися тим, що потрібно підтримувати лише шість арифметико-логічних операцій: 32-бітний XOR, додавання за модулем 2^{32} та обертання на 16, 17, 24 та 31 біт. Мінімалістичний 32-бітний арифметико-логічний блок (ALU) для цих шести операцій може бути добре оптимізований для досягнення невеликої площі кремнію та низького енергоспоживання.

Цілісність на всіх рівнях безпеки

Schwaemm та Esch були розроблені таким чином, щоб вони відповідали рівню безпеки, що полегшує параметризовану програмну реалізацію алгоритмів та базову перестановку. Усі екземпляри Schwaemm та Esch можуть використовувати одну реалізацію Sparkle, параметризовану щодо розміру блоку (тобто стану) та кількості кроків. Така параметризована реалізація значно скорочує зусилля з розробки програмного забезпечення, оскільки потрібно реалізувати та протестувати лише одну функцію для Sparkle.

Висока швидкість при паралельному запуску

Продуктивність Schwaemm та Esch на процесорних платформах з векторними двигунами (наприклад, ARM NEON, Intel SSE/AVX) можна значно збільшити, скориставшись паралелізацією рівня SIMD, яку вони надають, оскільки всі 32-бітні слова стану виконують ті самі операції в тому самому порядку. Реалізація обладнання може поміняти продуктивність на області кремнію шляхом застосування декількох 32-бітних ALU, які працюють паралельно.

Безпека sponge конструкцій

Захищеність цих схем ґрунтується на захисті криптографічних перестановок, що лежать в основі, та захищеності режимів на основі губки, точніше режиму хешування на основі губки та режиму Beetle для автентифікованого шифрування. Підхід на основі губки привернув багато уваги, оскільки він використовувався в останній стандартизованій хеш-функції, стандартизованій NIST, SHA-3. Ми повторно використовуємо цей підхід, щоб користуватися такими перевагами як

невеликий обсяг пам'яті, так і довірою криптографів до таких компонентів.

Література про розробку блокових шифрів

Конструкція перестановки сімейства Sparkle заснована на багаторічній структурі SPN, що дозволяє розкласти її аналіз на два етапи: по-перше, вивчення шару заміщення, а по-друге, вивчення шару лінійних операцій. Останній поєднує структуру Фейстеля, яка застосовується з моменту публікації DES, та лінійну перестановку з великим числом розгалуження, як і велика кількість SPN, таких як AES. Щоб поєднати ці два типи підкомпонентів, ми покладаємось на стратегію розробки, яка використовувалася для блочного шифру Sparx: стратегія довгого сліду. Наш шар заміщення працює на 64-бітних гілках, використовуючи ARX блоки. Дослідження диференціальних та лінійних властивостей модульного додавання в контексті блочного шифру можна простежити до кінця 90-х. Той факт, що розмір блоку компонента ARX обмежений 64 бітами, означає, що його можна ретельно дослідити за допомогою комп'ютерних методів. Простота та особливість форми лінійного шару дозволяє нам вивести властивості повної перестановки з властивостей 64-розрядного ARX-блока.

Компоненти з урахуванням випадків їх використання

При використанні перестановки в режимі роботи можливі два підходи. Ми можемо використовувати герметичний підхід, тобто немає відомих розрізнювачів проти перестановки. Потім ця безпека передається безпосередньо всій функції (наприклад, всій хеш-функції або схемі AEAD). Мінусом у цьому випадку є те, що ця герметична стратегія вимагає складної перестановки, яка в умовах легкої криптографії може

бути занадто важкою.

Навпаки, ми можемо використовувати перестановку, яка сама по собі не може забезпечити необхідні властивості. Потім захист забезпечується з'єднанням перестановки з режимом роботи, в якому вона використовується. Наприклад, нещодавно оголошений переможець конкурсу CAESAR ASCON та кандидат третього туру CAESAR Ketje, обидва аутентифіковані шифри, використовують такий підхід. Перевага в цьому випадку полягає в набагато більшій ефективності, оскільки нам потрібно менше раундів перестановки. Однак гарантії безпеки в цьому випадку апріорі слабкіші, оскільки складніше оцінити силу, необхідну перестановці.

Для Sparkle (і, таким чином, для Esch і Schwaemm) ми використовуємо останній підхід: перестановка, що використовується, має ряд раундів, які можуть дозволити існування деяких розрізнювачів. Однак, застосовуючи нову встановлену стратегію довгого сліду, можна довести, що наші алгоритми на основі губки є безпечними щодо найбільш важливих векторів диференціальних атак та лінійних атак із зручним запасом безпеки. У описі не згадується про можливості впроваджувати збої у процес роботи пристрою, зокрема у роботу процедури Sparkle, що робить шифр потенційно вразливим до атак збоїв, що будуть розглянуті нижче.

3.2 Автентифікований шифр SCHWAEMM

Schwaemm – це сімейство шифрів для автентифікованого шифрування із супутніми даними. Назва розшифровується Sponge-based Cipher for Hardened but Weightless Authenticated Encryption on Many Microcontrollers. Це також люксембурзьке слово «губки». Schwaemm використовує сімейство перестановок Sparkle в режимі роботи Beetle (який заснований на дуплексній губці). Розробники пропонують чотири екземпляри автентифікованого шифрування із супутніми даними, а саме

Schwaemm128-128, Schwaemm256-128, Schwaemm192-192 та Schwaemm256-256, які для заданого ключа K та нонсу N дозволяють обробляти пов'язані дані A та повідомлення M довільної довжини та виводити шифротекст C з $|C| = |M|$ та тегом автентифікації T. Для заданих (K, N, A, C, T) процедура дешифрування повертає розшифровку M шифртексту C, якщо T тег є валідним, інакше він повертає символ помилки $\perp$. Наш головний кандидат з сім'ї, який ми розглядаємо – Schwaemm192-192. Усі екземпляри використовують (із незначними змінами) режим роботи Beetle, який базується на добре відомій конструкції дуплексної губки. Різниця між екземплярами – це версія основної перестановки Sparkle (і, отже, швидкість і стійкість різна) і розмір тегу автентифікації. В якості угоди при іменуванні використовувалось $Schwaemmr - c$, де r посилається на розмір рейту і c до розміру ємності в бітах. Як і для хешування, використовується велика версія Sparkle для ініціалізації, проміжку між обробкою асоційованих даних та обробкою секретного повідомлення, і фіналізацією, і легка версія Sparkle для оновлення проміжного стану. У таблиці 3.1 наведено огляд параметрів екземплярів Schwaemm. Межі даних відповідають 2^{64} блокам по r біт, округлених до найближчого ступеня двох, за винятком Schwaemm256-256 з високою безпекою, для якого вони становлять $r * 2^{128}$ біт.

Таблиця 3.1 – Члени сімейства Schwaemm з відповідними рівнями безпеки в бітах з врахуванням кофіденційності цілісності та обмеженнями даних (у байтах) для обробки. Перший рядок відповідає основному кандидату

	n	r	c	$\|K\|$	$\|N\|$	$\|T\|$	security	data limit
Schwaemm192-192	384	192	192	192	192	192	184	2^{68}
Schwaemm256-128	384	256	128	128	256	128	120	2^{68}
Schwaemm128-128	256	128	128	128	128	128	120	2^{68}
Schwaemm256-256	512	256	256	256	256	256	248	2^{133}

3.2.1 Алгоритм

Основна відмінність режиму Beetle – це використання функції комбінованого зворотнього зв'язку ρ для диференціації блоків шифртексту та рейту частини станів. Цей комбінований зворотний зв'язок створюється за допомогою застосування функції $FeistelSwap$ до рейтової частини стану, яка обчислюється як

$$FeistelSwap(S) = S_2||(S_2 \oplus S_1),$$

де $S \in F_2^r$ та $S_1||S_2 = S$ з $|S_1| = |S_2| = \frac{r}{2}$. Функція зворотнього зв'язку $\rho : (F_2^r \times F_2^r) \to (F_2^r \times F_2^r)$ визначається як $\rho(S, D) = (\rho_1(S, D), \rho_2(S, D))$, де

$$\rho_1 : (S, D) \mapsto FeistelSwap(S) \oplus D, \rho_2 : (S, D) \mapsto S \oplus D.$$

Для дешифрування ми повинні використовувати функцію зворотнього зв'язку ρ' : $(F_2^r \times F_2^r)$, визначену як

$\rho'(S, D) = (\rho'_1(S, D), \rho'_2(S, D))$, де

$$\rho'_1 : (S, D) \mapsto FeistelSwap(S) \oplus S \oplus D, \rho'_2 : (S, D) \mapsto S \oplus D.$$

Після кожного застосування ρ та додавання раундових констант, тобто перед кожним викликом перестановки Sparkle, окрім ініціалізації, ми додаємо шар забілювання рейту, який XORить значення $W_{c,r}(S_R)$ до рейту, де S_R відповідний внутрішній стан, що відноситься до частини ємності. Для екземплярів Schwaemm з $r = c$ визначаємо $W_{c,r} : F_2^c \mapsto F_2^r$ (тобто, ми просто XORимо частину ємності до частини рейту). Для Schwaemm256-128 визначаємо $W_{128,256}(x, y) = (x, y, x, y)$, де $x, y \in F_2^{64}$. Зауважимо, що цей твік все ще можна описати у фреймворці Beetle, так як забілювання рейту можна вважати частиною визначення відповідної перестановки.

На малюнку 3.1 зображено режим для основного кандидата Schwaemm192-192. Формальний опис специфікації процедур шифрування та дешифрування наведені у Алгоритмах 3.1 та 3.2.

Схема атаки

Як відомо, атаки з фіксованими випадковими збоями застосовуються до останнього раунду перетворення, таким чином, щоб використовувати отримані збиті та коректні дані для побудови статистичних розпізнавачів. У нашому випадку, ми можемо розділити тег автентифікації на частини розміром з гілку відповідного ARX-блоку Sparkle, тобто на частини (x, y) по 64 біти, де $|x| = 32$ біти та $|y| = 32$ біти. Розглянуті нижче функції розпізнавання націлені на 3,4,5 гілки відповідно, перебор ключів відбувається для кожної гілки окремо.

Хоча шифр Ascon має схожу структуру губки, атака random-And не застосовна до даного шифру, оскільки там в якості шару заміщення використовуються 5-бітні S-блоки, що впливають на конкретні біти ключа

Algorithm 3.1 Процедура шифрування Schwaemm

Require: $(K, N, A, M) where K \in F_2^{192} is a key, N \in F_2^{192}$ is a nonce and $A, M \in F_2^*$
Ensure: $(C, T), where C \in F_2^*$ is a ciphertext and $T \in F_2^{192}$ is a authentication tag

if $A \neq \epsilon$ **then**
  $A_0||A_1||...||A_{l_A-1} \leftarrow A$ with $\forall i \in \{0, ..., l_A - 2\} : |A_i| = 192, 1 \leq |A_{l_A-1}| \leq 192$
  if $|A_{l_A-1}| < 192$ **then**
    $|A_{l_A-1}| \leftarrow pad_{192}(A_{l_A-1})$
    $Const_A \leftarrow 0 \oplus (1 \ll 3)$
  else
    $Const_A \leftarrow 1 \oplus (1 \ll 3)$
  end if
end if
if $M \neq \epsilon$ **then**
  $M_0||M_1||...||M_{l_M-1} \leftarrow A$ with $\forall i \in \{0, ..., l_M - 2\} : |M_i| = 192, 1 \leq |M_{l_M-1}| \leq 192$
  $t \leftarrow |M_{l_M-1}|$
  if $|M_{l_M-1}| < 192$ **then**
    $|M_{l_M-1}| \leftarrow pad_{192}(M_{l_M-1})$
    $Const_M \leftarrow 2 \oplus (1 \ll 3)$
  else
    $Const_M \leftarrow 3 \oplus (1 \ll 3)$
  end if
end if
$S_L||S_R \leftarrow Sparkle384_{11}(N||K) with |S_L| = 192, |S_R| = 192$
if $A \neq \epsilon$ **then**
  for all $j = 0, ..., l_A - 2$ **do**
    $S_L||S_R \leftarrow Sparkle384_7((\rho_1(S_L, A_j) \oplus S_R||S_R))$
  end for
  $S_L||S_R \leftarrow Sparkle384_11((\rho_1(S_L, A_{l_A-1}) \oplus S_R \oplus Const_A||(S_R \oplus Const_A))$
end if
if $M \neq \epsilon$ **then**
  for all $j = 0, ..., l_M - 2$ **do**
    $C_j \leftarrow \rho_2(S_L, M_j)$
    $S_L||S_R \leftarrow Sparkle384_7((\rho_1(S_L, M_j) \oplus S_R)||S_R)$
  end for
  $C_{l_M-1} \leftarrow trunc_t(\rho_2(S_L, M_{l_M-1}))$
  $S_L||S_R \leftarrow Sparkle384_11((\rho_1(S_L, M_{l_M-1}) \oplus S_R \oplus Const_M||(S_R \oplus Const_M))$
end if
return $(C_0||C_1||...||C_{l_M-1}, S_R \oplus K)$

Algorithm 3.2 Процедура розшифрування Schwaemm

Require: $(K, N, A, C, T) where K \in F_2^{192} is a key, N \in F_2^{192}$ is a nonce and $A, C \in F_2^*, T \in F_2^{192}$
Ensure: Decryption M of C if the tag T is valid, $\perp$ otherwise

if $A \neq \epsilon$ **then**
  $A_0||A_1||...||A_{l_A-1} \leftarrow A$ with $\forall i \in \{0, ..., l_A - 2\} : |A_i| = 192, 1 \leq |A_{l_A-1}| \leq 192$
  if $|A_{l_A-1}| < 192$ **then**
    $|A_{l_A-1}| \leftarrow pad_{192}(A_{l_A-1})$
    $Const_A \leftarrow 0 \oplus (1 \ll 3)$
  else
    $Const_A \leftarrow 1 \oplus (1 \ll 3)$
  end if
end if
if $C \neq \epsilon$ **then**
  $C_0||C_1||...||C_{l_M-1} \leftarrow C$ with $\forall i \in \{0, ..., l_M - 2\} : |C_i| = 192, 1 \leq |C_{l_M-1}| \leq 192$
  $t \leftarrow |C_{l_M-1}|$
  if $|C_{l_M-1}| < 192$ **then**
    $|C_{l_M-1}| \leftarrow pad_{192}(C_{l_M-1})$
    $Const_M \leftarrow 2 \oplus (1 \ll 3)$
  else
    $Const_M \leftarrow 3 \oplus (1 \ll 3)$
  end if
end if
$S_L||S_R \leftarrow Sparkle384_{11}(N||K) with |S_L| = 192, |S_R| = 192$
if $A \neq \epsilon$ **then**
  for all $j = 0, ..., l_A - 2$ **do**
    $S_L||S_R \leftarrow Sparkle384_7((\rho_1(S_L, A_j) \oplus S_R||S_R))$
  end for
  $S_L||S_R \leftarrow Sparkle384_11((\rho_1(S_L, A_{l_A-1}) \oplus S_R \oplus Const_A||(S_R \oplus Const_A))$
end if
if $C \neq \epsilon$ **then**
  for all $j = 0, ..., l_M - 2$ **do**
    $M_j \leftarrow \rho_2'(S_L, C_j)$
    $S_L||S_R \leftarrow Sparkle384_7((\rho_1'(S_L, C_j) \oplus S_R)||S_R)$
  end for
  $M_{l_M-1} \leftarrow trunc_t(\rho_2'(S_L, C_{l_M-1}))$
  if t < 192 **then**
    $S_L||S_R \leftarrow Sparkle384_11((\rho_1(S_L, pad_{192}(M_{l_M-1}) \oplus S_R \oplus Const_M||(S_R \oplus Const_M))$
  else
    $S_L||S_R \leftarrow Sparkle384_11((\rho_1'(S_L, C_{l_M-1}) \oplus S_R \oplus Const_M||(S_R \oplus Const_M))$
  end if
end if
if $S_R \oplus K = T$ **then**
  return $(M_0||M_1||...||M_{l_M-1})$
else
  return $\perp$
end if

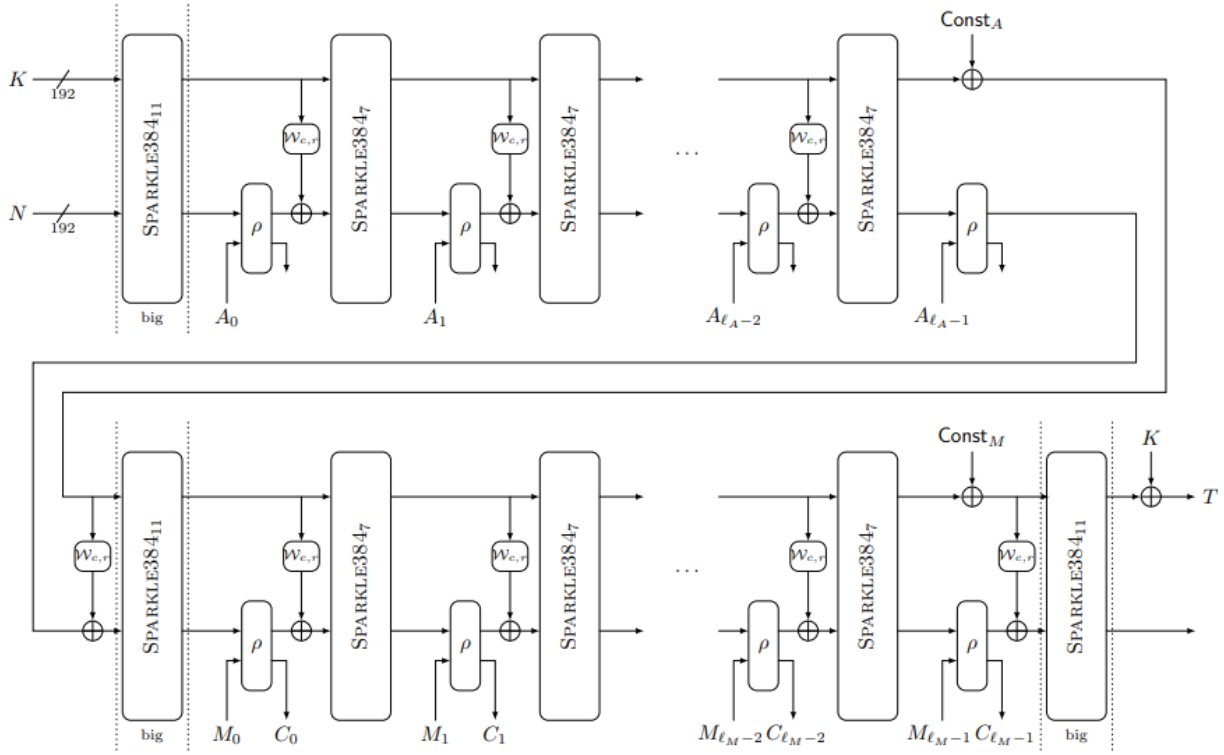

Рисунок 3.1 – Автентифікований алгоритм шифрування Schwaemm192-192 з $r = 192, c = 192$

під час операції XOR, проте у випадку шифру Schwaemm, один раунд заміщення для двох слів використовує 64 біти ключа, що не дає можливості розбивати ключ на менші частини для зменшення кількості перебору. На малюнку 3.2 зображені ARX-блоки, які беруть участь в операції XOR з ключем.

Сутність атаки полягає у внесенні помилки в оброблюваний блок даних безпосередньо перед останнім раундом функції Sparkle та порівнянні одержаного «збитого» значення тегу T із оригінальним за допомогою функцій розпізнавання [2, 5]. Оскільки ключ у 192 біти занадто великий для прямого перебору, ми розділимо його на w частин (w ділить 192) довжини $192/w$. Нехай у нашому випадку $w = 6$, тоді маємо функції розпізнавання для помилки у збитих блоках, де $K_i \in \{K_0..K_4\}$, $T_i \in \{T_0..T_4\}$:

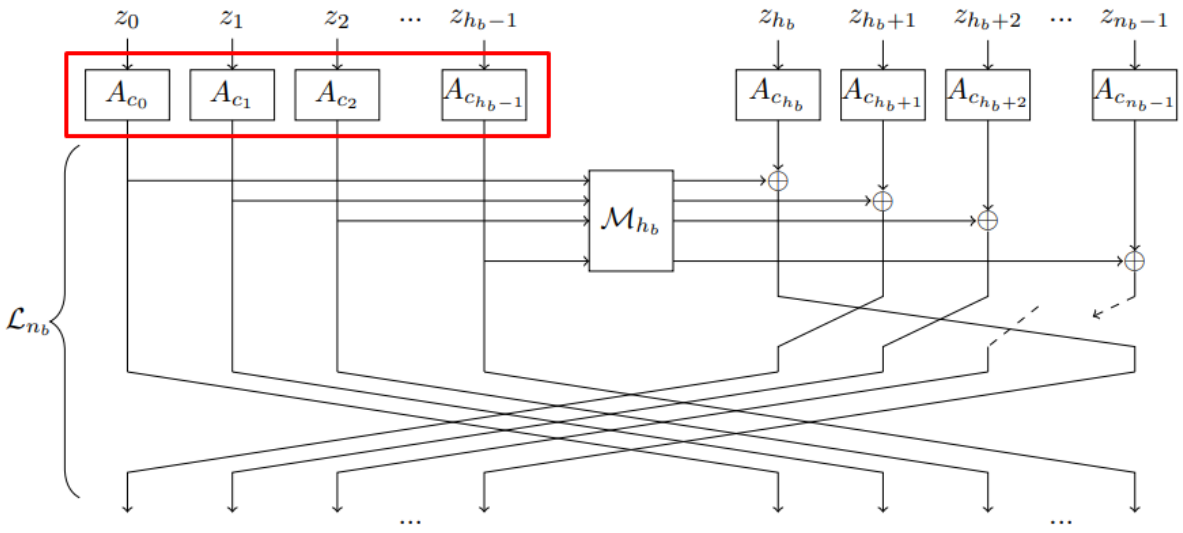

Рисунок 3.2 – Схема раунду процедури SPARKLE, де z_i – це 64-бітні гілки, які подаються на вхід відповідним ARX-блокам, червоним позначені блоки, в яких атакуємо проміжне значення перед останнім раундом

$$g(K_i, K_{i+1}) = [(\tilde{T}_i \oplus K_i \oplus c) - ((((\tilde{T}_i \oplus K_i \oplus c) >>> 16) \oplus$$

$$\oplus(\tilde{T_{i+1}} \oplus K_{i+1})) >>> 24)] \oplus$$

$$\oplus[(T_i \oplus K_i \oplus c) - ((((T_i \oplus K_i \oplus c) >>> 16)$$

$$\oplus(T_{i+1} \oplus K_{i+1})) >>> 24)],$$

де $\tilde{T}_i$ – «збитий» тег, T_i – оригінальний тег.

Вигляд цієї функції обумовлений структурою останнього раунду процедури Sparkle У роботі розглядається випадкова модель збоїв, де кожен збій розміром 32 біти, тобто змінюємо значення одного з 6-ти 32-бітних проміжних слів, які беруть участь в операції $\oplus$ з ключем, на випадкове.

Враховуючи внутрішню структуру шифру Sparkle (Алгоритм 3.3), маємо вносити збій в слова $Z_{i,0}$, $Z_{i,1}$, $Z_{i,2}$ останнього раунду, оскільки $Z_{i,3}$,

$Z_{i,4}$, $Z_{i,5}$ не впливають на ту частину, яка бере участь в операції $\oplus$ з ключем. Ми припускаємо, що вміємо запускати процедуру шифрування з фіксованим ключем K та фіксованим значенням nonce N, що загалом припустимо для подібних атак.

Algorithm 3.3 ARX-блок A_c

Require: $(x, y) \in F_2^{32} \times F_2^{32}$
$x \leftarrow x + (y >>> 31)$
$y \leftarrow y \oplus (x >>> 24)$
$x \leftarrow x \oplus c$
$x \leftarrow x + (y >>> 17)$
$y \leftarrow y \oplus (x >>> 17)$
$x \leftarrow x \oplus c$
$x \leftarrow x + (y >>> 0)$
$y \leftarrow y \oplus (x >>> 31)$
$x \leftarrow x \oplus c$
$x \leftarrow x + (y >>> 24)$
$y \leftarrow y \oplus (x >>> 16)$
$x \leftarrow x \oplus c$
return $(x, y) \in F_2^{32} \times F_2^{32}$

Алгоритм виглядає таким чином:

1) згенерувати R тегів перевірки та виконати R збоїв в останньому раунді фінального перемішування функції Sparkle (випадковий збій у фіксованому блоці під час кожного шифрування);

2) для кожної частини-кандидата ключа K_i порахувати розподіл значень функцій розпізнавання $g^{(j)}$, де j – номер тексту;

3) перевірити гіпотезу «$g^{(j)}$ має розподіл, що максимально відрізняється від рівноймовірного».

Розрізняти гіпотези будемо за допомогою евклідової відстані за формулою

$$d(K_i) = \sum_{x=0}^{2^{192/w}} \left(\frac{\#(g^{(j)}(K_i) = x)}{R} - \frac{1}{2^{192/w}} \right)^2,$$

де $\#(g^{(j)}(K_i) = x)$ означає кількість значень по всіх тегах автентифікації при фіксованих K_i. Розподіл значень припускаючих функцій $g^{(j)}$ залежить від розподілу внесених у R тегів помилок і очікується близьким до рівномірного, якщо помилки у словах $Z_{i,0}$, $Z_{i,1}$, $Z_{i,2}$ немає, і нерівномірним, якщо є.

3.3 Експериментальна перевірка

Для практичної перевірки гіпотези було побудовано модель шифру Schwaemm на мові програмування Java, використовуючи функцію SecureRandom для генерування випадкових значень помилки. Далі для кожної з 2^{64} частини ключа обчислюємо значення $d(K_i)$ для 1000 або більше шифротекстів (звичайних та збитих).

Відповідні максимальній евклідовій відстані значення і є нашими кандидатами в ключі. Проте у цьому випадку перебір все ж не є практичним для пересічного користувача, ми зменшили перебір ключа до 2^{64}, проте це все ж недостатньо для практичного проведення атаки через внутрішню структуру шифру Sparkle, у якому для проведення зворотньої операції все ж необхідно знати 64 біти ключа (K_i, K_{i+1}) (по 32 біти на кожне слово). Для ключа довжини 192 біти маємо повторити цю атаку 3 рази відповідно, щоб відновити повний ключ шифрування.

Контрзаходи

Контрзаходи проти атак можуть бути класифіковані на три загальні категорії:

1) виявлення помилок;

2) випадковість помилок;

3) контрзаходи на основі сенсорів.

Методи виявлення збоїв використовують в роботі шифру надмірності для виявлення помилок та вживають відповідних заходів для придушення несправних значень. Приклади такої надлишкової арифметики для виявлення помилок в роботі шифрів включають [18] для 8-бітових S-блоків, [19] для легких шифрів, та [20] використання надлишковості програмного забезпечення, що називається внутрішніми контрзаходами надлишковості (IRC), використовуючи інструкції SIMD для блокового

шифру PRIDE та потокового шифру TRIVIUM.

Другий клас контрзаходів, тобто рандомізація помилки, реалізує додаткові операції в шифрі таким чином, що помилка, викликана введенням збоїв рандомізується та корисна інформація про залежний від даних розподіл не просочується до нападника. Приклад включає контрзахід на AES, запропонований в [21] та оцінений в [22]. У цій техніці стан шифру порівнюється із надлишковим станом і, у випадку різниці, що виникає внаслідок помилок, стан замінюється підставним станом. Отже, замість збитого стану, випадковий стан піддається впливу зловмисника. Щоб запобігти тому, що зловмисник може вчинити такий самий збій у шифрі, і надлишковому стані [20] пропонують трансформувати простір перетворення збоїв, при якому шифрувальні та надлишкові операції виконуються під різними кодуваннями. В результаті простір збоїв у операції шифрування відображається на інший простір збоїв у надлишкових операціях. Відображення обирається таким чином, щоб розподіл збитих значень у шифрі та надлишкових станах був різним. Усі вищезазначені контрзаходи зосереджені на тому випадку, коли помилки виникають при роботі шифру під час впровадження збою і намагаються запобігти передачі будь-якої інформації про розподіл помилки нападнику. У техніці аналізу збоїв на основі введення неефективних збоїв, тільки коректні виходи необхідні для вилучення секретних ключів. Звідси такі контрзаходи, як обговорено вище, не захищають реалізацію від неефективних збоїв. Нормальні контрзаходи для атак збоїв, що використовують неефективні збої, включають також виявлення введених збоїв, незалежно від їх впливу на оброблювані дані, або видаляють залежність даних від розподілу помилки.

Третій клас контрзаходів проти атак збоїв - це методи, засновані на сенсорах, які "відчувають"механізм введення збою незалежно від того, чи викликає він помилку в роботі шифру. Вставляються відповідні датчики в реалізацію шифру для виявлення механізмів введення збоїв, наприклад, температурні удари, світло або електромагнітне випромінювання, і

аномальні зсуви живлення/тактів виконання [20] - [25]. Обмеження цих типів контрзаходів – це різноманітність механізмів введення помилок. Універсальний датчик, який може виявити всі можливі механізми, не існує.

В альтернативному класі контрзаходів проти аналізу розподілу, проміжні змінні роблять випадковими так, що розподіл збоїв більше не залежить від значення секрету. Маскування – поширений контрзахід, що застосовується проти диференціального аналізу споживання (DPA), оскільки кореляція між витратою енергії та даними зменшується використовуючи рандомізовані проміжні значення [22], [23]. Контрзаходи проти комбінованого аналізу споживання/збоїв включають коди управління помилками (ECC); в масковану реалізацію для зменшення витрат на захист шифру від різних типів атак [24], [25]. Однак щодо захисту від атак збоїв, такі методи можна класифікувати за контрзаходами виявлення збоїв, проти яких не ефективний аналіз неефективних збоїв. Дослідження [26] показують, що контрзаходи виявлення збоїв на основі паритету/залишку перевіряють схему полегшення атак на споживання з посиленою вигодою нападника у кількості контрольних бітів.

Основна властивість маскування, що захищає реалізацію проти аналізу споживання рандомізує проміжні змінні, так, що немає кореляції між споживанням енергії та таємним значенням. Аналогічно до захисту від упереджених атак збоїв, маскування може декорелювати вірогідність помилки через введення збою та секретне значення. Однак якщо зловмисник здатний викликати збої в усіх блоках, сума всіх блоків все ще є розподіленою та атака залишається успішню [5]. Отже, хоча маскування може зробити цю атаку складнішою, вимагаючи від зловмисника вчинити збої у кількох місцях одночасно, все ще можливо атакувати масковану імплементацію з введенням розподілених певним чином збоїв. Однак ідею генерування випадкових незалежних блоків у звичайних методах маскування можна адаптувати для розробки контрзаходу, що призводить

до рівномірного розподілу проміжних змінних навіть при введенні розподілених певним чином збоїв. Потрібно більше досліджень для аналізу ефективності таких методик маскування проти аналізу неефективних збоїв.

Висновки до розділу 3

У даному розділі було проведено дослідження стійкості кандидату з сімейства AEAD шифрування Schwaemm192-192 до атак збоїв. Вдалось сформулювати модель побудови даного типу атак у запропонованих обмеженнях. Ці обмеження є достатніми для зменшення перебору ключового простору до 2^{64}, використовуючи три пристрої одночасно. Ми розділяємо ключ на 3 частини довжини 64 біти, як і довжина гілки Sparkle блоку, та використовуючи зворотні перетворення за допомогою функцій розпізнавання відновлюємо значення збою внутрішнього стану Sparkle, що дозволяє аналізувати його розподіл. Додаткові збої або використання інших побічних каналів необхідно для практичної атаки. Для того, щоб покращити алгоритм знаходження ключів, криптоаналітику потрібно вносити не випадкові, а конкретні помилки, що дозволить контролювати розподіл значень $g^{(j)}$ і користуватися максимальною евклідовою відстанню ефективніше. Запропонована модель атаки на Ascon може бути застосовна у якості подвійних збоїв, проте не вдається зменшити ключ більше ніж на 64 біти, оскільки ці частини повністю беруть участь у ARX-конструкції, на відміну від Ascon, де використовуються 5-бітні S-блоки.

Для захисту від атак збоїв, де помилка належить до певного розподілу на практиці використовуються методи знаходження помилок або маскування проміжних значень, що може декорелювати вірогідність помилки.

ВИСНОВКИ

Атаки збоїв можна застосувати до будь-якого шифру, побудованого на схемі Фейстеля, зокрема до ARX-блоку Sparkle в автентифікованому шифрі Schwaemm, що й було основною темою мого дослідження. Як можна побачити, ані атаки Біхама-Шаміра, ані атаки Рівайна не використовували внутрішню структуру раундового перетворення DES. Однак атака на останній раунд процедури Sparkle, описана в останньому розділі, використовувала обернену раундову функцію. Було виявлено, що шифр піддається атакам заміни операцій під час фінального додавання із ключем та класичним атакам збоїв конкретних чи випадкових слів під час раундового перетворення. Також були запропоновані розпізнаючі функції для збору статистичних даних на основі евклідової відстані. Втім, дані атаки відновлюють не власне ключ шифрування, а лише одну з його частин, відповідно до структури ARX-блоку, у шифрі Schwaemm це два слова по 32 біти. Для визначення інших частин ключа та відновлення вихідного ключа шифрування потрібно проводити атаку ще таку кількість разів, на скільки частин був розділений ключ, в нашому випадку 3 рази відповідно. Стратегія, описана в роботі, дозволяє проводити атаку на 3 пристроях одночасно, з різними частинами ключа. Хоча повний перебір розміру 2^{64} все ж менше, ніж перебір всього ключа розміру 2^{192}, проте є занадто великим для практичного проведення атаки на пересічній машині. Даний підхід вимагає додаткових технік для реалізації плану.

Наявність блокової структури у раундовій функції дуже спрощує аналіз, оскільки для блокової структури збої можна вносити в окремі блоки, що локалізує помилку і дозволяє аналізувати розподіл на меншій кількості даних, що зменшує розмір перебору ключа. В той же час відсутність блокової структури не є перешкодою для наведених атак: схеми статистичного розпізнавання не змінюються. Однак обчислювальна

складність в цьому випадку суттєво зростає за рахунок великої кількості значень раундового ключа. Тому для шифрів, раундові функції яких не мають блокової структури, перед проведенням даних атак треба додатково оцінити загальну складність визначення ключа шифрування (чи деякої його частини) у порівнянні із простим перебором. Враховуючи викладені думки та огляди атак на схожі шифри, зокрема Ascon, є досить надійним шифром та можливо переможе у конкурсі Lightweight Cryptography Standartization Process.

ПЕРЕЛІК ПОСИЛАНЬ

1. D. J. Bernstein (2016) Cryptographic competitions. [Online]. Available: https://competitions.cr.yp.to/caesar.html

2. P. Kocher, J. Jaffe, B. Jun, and P. Rohatgi, “Introduction to differential power analysis”, Journal of Cryptographic Engineering, vol. 1, no. 1, pp. 5–27, 2011.

3. Christof Beierle, Alex Biryukov, Luan Cardoso dos Santos, Johann Großschadl, L’eo Perrin, Aleksei Udovenko, Vesselin Velichkov, and Qingju Wang, "Schwaemm and Esch: Lightweight Authenticated Encryption and Hashing using the Sparkle Permutation Family Available: https://csrc.nist.gov/CSRC/media/Projects/Lightweight-Cryptography/documents/round-1/spec-doc/SPARKLE-spec.pdf

4. E. Ronen, A. Shamir, A.-O. Weingarten, and C. OFlynn, “IoT goes nuclear: Creating a ZigBee chain reaction,” in Security and Privacy (SP), 2017 IEEE Symposium on, 2017, pp. 195–212.

5. C. Dobraunig, M. Eichlseder, T. Korak, S. Mangard, F. Mendel, R. Primas, “SIFA: Exploiting Ineffective Fault Inductions on Symmetric Cryptography,” pp. 547–572, 2018.

6. Keyvan Ramezanpour, Paul Ampadu, and William Diehl, A Statistical Fault Analysis Methodology for the Ascon Authenticated Cipher, Available: $https://rijndael.ece.vt.edu/wdiehl/Keyvan_HOST_2019.pdf$

7. R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks, and L. Wingers, The SIMON and SPECK lightweight block ciphers, in Proceedings of the 52 Annual Design Automation Conference, San Francisco, CA, USA, June 7-11, 2015, 2015, pp. 175:1–175:6.

8. J.P. Aumasson, L. Henzen, W. Meier, and C.W. Phan, Sha-3 proposal blake, Submission to NIST, 2008.

9. D. Naccache, Finding faults [data security], IEEE Security & Privacy, vol. 3, no. 5, pp. 61–65, 2005.

10. J. Blomer and J.P. Seifert, Fault based cryptanalysis of the advanced encryption standard (aes), in Computer Aided Verification. Springer, 2003, pp. 162–181.

11. M. Tunstall, D. Mukhopadhyay, and S. Ali, Differential fault analysis of the advanced encryption standard using a single fault, in IFIP International Workshop on Information Security Theory and Practices. Springer, 2011, pp. 224–233.

12. Eli Biham, Adi Shamir. Differential fault analysis of secret key cryptosystems. LNCS 1294/1997. pp. 513-525. Available: https://pdfs.semanticscholar.org/440f/a56b0618578b34c7a4fb781fc40388bf8e18.pdf.

13. Matthieu Rivain, Differential Fault Analysis on DES Middle Rounds. LNCS 5747/2009. pp. 457–469. Available: https://www.iacr.org/archive/ches2009/57470460/57470460.pdf.

14. D. Naccache. Finding faults [data security]. IEEE Security and Privacy. vol. 3, no. 5, pp. 61–65, 2005.

15. Dilip Kumar, Sikhar Patranabis, Jakub Breier. A Practical Fault Attack on ARX-like Ciphers with a Case Study on ChaCha20. SEAL, Department of Computer Science and Engineering, IIT Kharagpur, India. 2017. Available: https://eprint.iacr.org/2017/1074.pdf.

16. H. Tupsamudre, S. Bisht, and D. Mukhopadhyay, "Destroying fault invariant with randomization," in International Workshop on Cryptographic Hardware and Embedded Systems. Springer, 2014, pp. 93–111.

17. S. Patranabis, A. Chakraborty, and D. Mukhopadhyay, "Fault tolerant infective countermeasure for AES," Journal of Hardware and Systems Security, vol. 1, no. 1, pp. 3–17, 2017.

18. S. Patranabis, A. Chakraborty, D. Mukhopadhyay, and P. P. Chakrabarti, "Fault space transformation: A generic approach to counter differential fault analysis and differential fault intensity analysis on AES-like block ciphers," IEEE Transactions on Information Forensics and Security, vol. 12, no. 5, pp. 1092–1102, 2017.

19. K. M. Zick, M. Srivastav, W. Zhang, and M. French, "Sensing

nanosecond-scale voltage attacks and natural transients in FPGAs," in Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays. ACM, 2013, pp. 101–104.

20. B. Selmke, S. Brummer, J. Heyszl, and G. Sigl, "Precise laser fault injections into 90 nm and 45 nm sram-cells," in International Conference on Smart Card Research and Advanced Applications. Springer, 2015, pp. 193–205.

21. W. He, J. Breier, S. Bhasin, N. Miura, and M. Nagata, "Ring Oscillator under Laser: Potential of PLL-based Countermeasure against Laser Fault Injection," in Fault Diagnosis and Tolerance in Cryptography (FDTC), 2016 Workshop on. IEEE, 2016, pp. 102–113.

22. T. De Cnudde, O. Reparaz, B. Bilgin, S. Nikova, V. Nikov, and V. Rijmen, "Masking AES with d + 1 Shares in Hardware," in International Conference on Cryptographic Hardware and Embedded Systems. Springer, 2016, pp. 194–212.

23. A. Adomnicai, J. J. Fournier, and L. Masson, "Masking the Lightweight Authenticated Ciphers ACORN and Ascon in Software," Cryptology ePrint Archive, Report 2018/708, 2018, https://eprint.iacr.org/2018/708.

24. T. Schneider, A. Moradi, and T. Guneysu, "ParTI–towards combined hardware countermeasures against side-channel and fault-injection attacks," in Annual International Cryptology Conference. Springer, 2016, pp. 302–332.

25. J. Dofe, H. Pahlevanzadeh, and Q. Yu, "A comprehensive FPGA-based assessment on fault-resistant AES against correlation power analysis attack," Journal of Electronic Testing, vol. 32, no. 5, pp. 611–624, 2016.

26. F. Regazzoni, T. Eisenbarth, L. Breveglieri, P. Ienne, and I. Koren, "Can knowledge regarding the presence of countermeasures against fault attacks simplify power attacks on cryptographic devices?" in 2008 IEEE International Symposium on Defect and Fault Tolerance of VLSI Systems. IEEE, 2008, pp. 202–210.

ДОДАТОК А ТЕКСТИ ПРОГРАМ

Тексти інструментальної програми для проведення експериментальних досліджень, що необхідно виносити у додатки.

A.1 Sparkle

```
import java.nio.ByteBuffer;
import java.nio.IntBuffer;
import java.util.ArrayList;
import java.util.Random;

public class GlobalMembers
{
        private static int ROT(int x, int n) {
                return Integer.rotateRight(x,n);
        }

        private static int ELL(int x)
        {
                return ROT((x ^ (x << 16)), 16);
        }

        public static final int[] RCON = { 0xB7E15162, 0xBF715
        private static final int MAX_BRANCHES = RCON.length;

        private static ArrayList<Integer> generateMassiveOfPla
                ArrayList<Integer> resultArray = new ArrayList
                for (int i = 0; i < numberOfInts ; i++) {
                        resultArray.add(new Random().nextInt()
```

```
        }
        return resultArray;
}

public static state_t linear_layer(state_t state, int
{
  int i;
  int b = nb / 2;
  int[] x = state.x;
  int[] y = state.y;
  int tmp;

  // Feistel function (adding to y part)
  tmp = 0;
  for (i = 0; i < b; i++)
  {
        tmp ^= x[i];
  }
  tmp = ELL(tmp);
  for (i = 0; i < b; i++)
  {
        y[i + b] ^= (tmp ^ y[i]);
  }

  // Feistel function (adding to x part)
  tmp = 0;
  for (i = 0; i < b; i++)
  {
        tmp ^= y[i];
  }
        tmp = ELL(tmp);
```

```
    for (i = 0; i < b; i++)
    {
            x[i + b] ^= (tmp ^ x[i]);
    }

    // Branch swap of the x part
    tmp = x[0];
    for (i = 0; i < b - 1; i++)
    {
            x[i] = x[i + b + 1];
            x[i + b + 1] = x[i + 1];
    }
    x[b - 1] = x[b];
    x[b] = tmp;

    // Branch swap of the y part
    tmp = y[0];
    for (i = 0; i < b - 1; i++)
    {
            y[i] = y[i + b + 1];
            y[i + b + 1] = y[i + 1];
    }
    y[b - 1] = y[b];
    y[b] = tmp;
    state.x = x;
    state.y = y;
    return state;
}

public static state_t linear_layer_with_error(state_t
{
```

```
int i;
int b = nb / 2;
int[] x = state.x;
int[] y = state.y;
int tmp;

// Feistel function (adding to y part)
tmp = 0;
for (i = 0; i < b; i++)
{
        tmp ^= x[i];
}
tmp = ELL(tmp);
for (i = 0; i < b; i++)
{
        y[i + b] ^= (tmp ^ y[i]);
}

// Feistel function (adding to x part)
tmp = 0;
for (i = 0; i < b; i++)
{
        tmp ^= y[i];
}
tmp = ELL(tmp);
for (i = 0; i < b; i++)
{
        x[i + b] ^= (tmp ^ x[i]);
}

// Branch swap of the x part
```

```
            tmp = x[0];
            for (i = 0; i < b - 1; i++)
            {
                    x[i] = x[i + b + 1];
                    x[i + b + 1] = x[i + 1];
            }
            x[b - 1] = x[b];
            x[b] = tmp;

            // Branch swap of the y part
            tmp = y[0];
            for (i = 0; i < b - 1; i++)
            {
                    y[i] = y[i + b + 1];
                    y[i + b + 1] = y[i + 1];
            }
            y[b - 1] = y[b];
            y[b] = tmp;
            state.x = x;
            state.y = y;
            return state;
    }


    private static state_t sparkle_ref(state_t state, int r
    {
            int[] x = state.x;
            int[] y = state.y;

            // The number of branches (nb) must be even and
            assert ((nb & 1) == 0) && (nb >= 4) && (nb <= M
```

```
            for (int i = 0; i < ns; i++)
            {
                    // Add step counter
                    y[0] ^= RCON[i % MAX_BRANCHES];
                    y[1] ^= i;
                    // ARXBox layer
                    for (int j = 0; j < nb; j++)
                    {
                            x[j] += ROT(y[j], 31);
                            y[j] ^= ROT(x[j], 24);
                            x[j] ^= RCON[j];
                            x[j] += ROT(y[j], 17);
                            y[j] ^= ROT(x[j], 17);
                            x[j] ^= RCON[j];
                            x[j] += y[j];
                            y[j] ^= ROT(x[j], 31);
                            x[j] ^= RCON[j];
                            x[j] += ROT(y[j], 24);
                            y[j] ^= ROT(x[j], 16);
                            x[j] ^= RCON[j];
                    }
                    // Linear layer
                    state = linear_layer(state, nb);
            }
            return state;
    }

    public static state_t linear_layer_inv(state_t state,
    {
      int i;
```

```
int b = nb / 2;
int[] x = state.x;
int[] y = state.y;
int tmp;

// Branch swap of the x part
tmp = x[b - 1];
for (i = b - 1; i > 0; i--)
{
        x[i] = x[i + b];
        x[i + b] = x[i - 1];
}
x[0] = x[b];
x[b] = tmp;

// Branch swap of the y part
tmp = y[b - 1];
for (i = b - 1; i > 0; i--)
{
        y[i] = y[i + b];
        y[i + b] = y[i - 1];
}
y[0] = y[b];
y[b] = tmp;

// Feistel function (adding to x part)
tmp = 0;
for (i = 0; i < b; i++)
{
        tmp ^= y[i];
}
```

```
    tmp = ELL(tmp);
    for (i = 0; i < b; i++)
    {
            x[i + b] ^= (tmp ^ x[i]);
    }

    // Feistel function (adding to y part)
    tmp = 0;
    for (i = 0; i < b; i++)
    {
            tmp ^= x[i];
    }
    tmp = ELL(tmp);
    for (i = 0; i < b; i++)
    {
            y[i + b] ^= (tmp ^ y[i]);
    }
    state.x = x;
    state.y = y;
    return state;
}


public static state_t sparkle_inv_ref(state_t state, in
{
            int[] x = state.x;
            int[] y = state.y;
            int i; // Step and branch counter
            int j;

              // The number of branches (nb) must be even a
```

```
        assert ((nb & 1) == 0) && (nb >= 4) && (nb <=

        for (i = ns - 1; i >= 0; i--)
        {
            // Linear layer
            state = linear_layer_inv(state, nb);
            // ARXbox layer
            for (j = 0; j < nb; j++)
            {
                x[j] ^= RCON[j];
                y[j] ^= ROT(x[j], 16);
                x[j] -= ROT(y[j], 24);
                x[j] ^= RCON[j];
                y[j] ^= ROT(x[j], 31);
                x[j] -= y[j];
                x[j] ^= RCON[j];
                y[j] ^= ROT(x[j], 17);
                x[j] -= ROT(y[j], 17);
                x[j] ^= RCON[j];
                y[j] ^= ROT(x[j], 24);
                x[j] -= ROT(y[j], 31);
            }
            // Add step counter
            state.y[1] ^= i;
            state.y[0] ^= RCON[i % MAX_BRANCHES];
    }
        state.x = x;
        state.y = y;
        return state;
}
```

```
public static void print_state_ref(state_t state, int nb)
                {
                        ByteBuffer byteBuffer = ByteBuffer.allo
                        IntBuffer intBuffer = byteBuffer.asIntB
                        intBuffer.put(state.x);
                        byte[] xbytes = byteBuffer.array();

                        ByteBuffer byteBuffer1 = ByteBuffer.all
                        IntBuffer intBuffer1 = byteBuffer1.asIn
                        intBuffer1.put(state.y);
                        byte[] ybytes = byteBuffer1.array();

                for (int i = 0; i < nb; i++) {
                        int j = 4 * i;
                        System.out.printf("(%02x%02x%02x%02x %
                                        xbytes[j], xbytes[j +
                                        ybytes[j], ybytes[j +
                        if (i < nb - 1) {
                        System.out.print(" ");
                        }
                }
                System.out.print("\n");
                }

        public static void main(String[] args)
        {
                state_t defaultState = new state_t(new int[] {0
                                new int[] {0xB7E15162, 0xBF715

                ArrayList<Integer> massiveOfTexts = generateMa
                System.out.println(massiveOfTexts);
```

```
        int nb = 6;
        int ns = 7;
        // 6 32-bit words for SPARKLE384 with 6 number
        state_t state = new state_t(new int[] {0xB7E15

        int[] message = new int[]{0xB7E15163, 0xBF7158

        System.out.print("input:\n");
        print_state_ref(state, nb);

        System.out.print("sparkle:\n");
        print_state_ref(sparkle_ref(state, nb, ns), nb
        //System.out.print("sparkle with errors:\n");
        //print_state_ref(sparkle_ref_withErrors(defau

        System.out.print("sparkle inv:\n");
        print_state_ref(sparkle_inv_ref(state, nb, ns)
//      System.out.print("sparkle inv with errors:\n")
//      print_state_ref(sparkle_inv_ref(defaultState, n
        System.out.print("\n");
    }

    private static state_t sparkle_ref_withErrors(state_t s

        int[] x = state.x;
        int[] y = state.y;
        int error = new Random().nextInt();
        // The number of branches (nb) must be even and
        assert ((nb & 1) == 0) && (nb >= 4) && (nb <= M
        for (int i = 0; i < ns-1; i++)
```

```
                {
                        // Add step counter
                        y[0] ^= RCON[i % MAX_BRANCHES];
                        y[1] ^= i;
                        // ARXBox layer
                        for (int j = 0; j < nb; j++)
                        {
                                x[j] += ROT(y[j], 31);
                                y[j] ^= ROT(x[j], 24);
                                x[j] ^= RCON[j];
                                x[j] += ROT(y[j], 17);
                                y[j] ^= ROT(x[j], 17);
                                x[j] ^= RCON[j];
                                x[j] += y[j];
                                y[j] ^= ROT(x[j], 31);
                                x[j] ^= RCON[j];
                                x[j] += ROT(y[j], 24);
                                y[j] ^= ROT(x[j], 16);
                                x[j] ^= RCON[j];
                        }
                        // Linear layer
                        state = linear_layer(state, nb);
                }
//last step
                y[0] ^= RCON[(ns-1) % MAX_BRANCHES];
                y[1] ^= (ns-1);
                // ARXBox layer
                for (int j = 0; j < nb-1; j++)
                {
                        x[j] += ROT(y[j], 31);
                        y[j] ^= ROT(x[j], 24);
```

```
                x[j] ^= RCON[j];
                x[j] += ROT(y[j], 17);
                y[j] ^= ROT(x[j], 17);
                x[j] ^= RCON[j];
                x[j] += y[j];
                y[j] ^= ROT(x[j], 31);
                x[j] ^= RCON[j];
                x[j] += ROT(y[j], 24);
                y[j] ^= ROT(x[j], 16);
                x[j] ^= RCON[j] ;
            }
            //last branch
            x[nb-1] += ROT(y[nb-1], 31);
            y[nb-1] ^= ROT(x[nb-1], 24);
            x[nb-1] ^= RCON[nb-1];
            x[nb-1] += ROT(y[nb-1], 17);
            y[nb-1] ^= ROT(x[nb-1], 17);
            x[nb-1] ^= RCON[nb-1];
            x[nb-1] += y[nb-1];
            y[nb-1] ^= ROT(x[nb-1], 31);
            x[nb-1] ^= RCON[nb-1];
            x[nb-1] += ROT(y[nb-1], 24);
            y[nb-1] ^= ROT(x[nb-1], 16);
            x[nb-1] ^= RCON[nb-1];
            // Linear layer
            state = linear_layer(state, nb);
            return state;
        }
    }
```